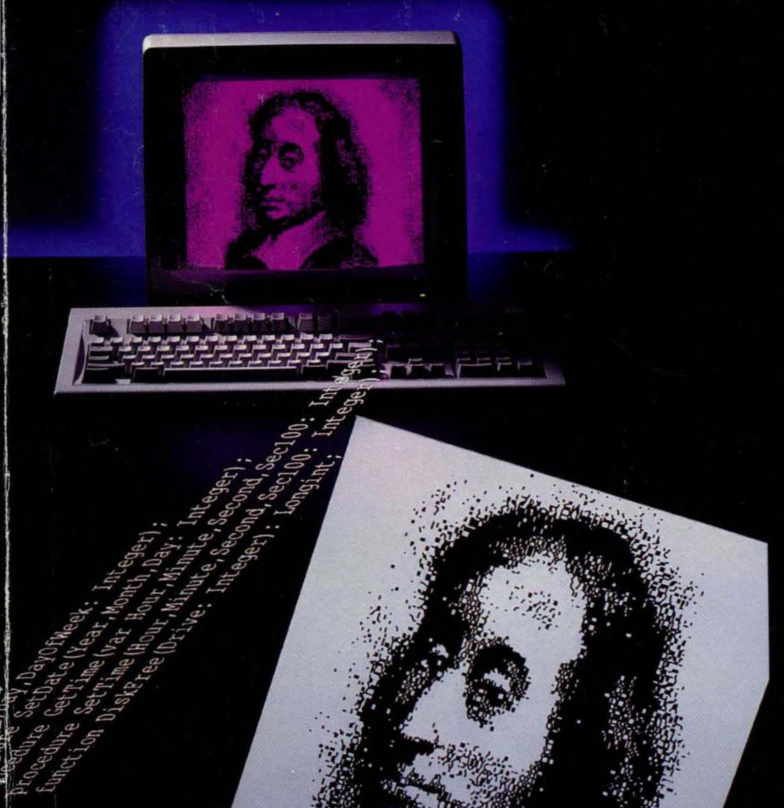
TURBO PASCAL®
GRAPHIX TOOLBOX

An essential collection of business graphics routines for your Turbo Pascal programs



Version 4.0

IBM® VERSION

PC, XT,® AT,® PS/2™ & True Compatibles

BORLAND

Turbo Pascal Graphix Toolbox®

Owner's Handbook

Version 4.0

BORLAND INTERNATIONAL, INC.
4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CALIFORNIA 95066

10 9 8 7 6 5 4 3 2 1

Table of Contents

Introduction

Welcome to the Turbo Pascal Graphix Toolbox. The procedures and functions that make up this software package will expand your repertoire of Turbo Pascal programming tools. With the aid of the Graphix Toolbox, you can develop high-resolution monochrome graphics for IBM PC and PC-compatible computers (using either an IBM CGA, EGA or 3270, AT&T 6300, or a Hercules graphics card).

This manual makes extensive use of Turbo Pascal programming examples; a good working knowledge of Turbo Pascal is assumed. If you need to brush up on your Pascal knowledge, refer to the *Turbo Pascal* manual, and/or the *Turbo Pascal Tutor*.

What Can You Do With the Graphix Toolbox?

The Turbo Pascal Graphix Toolbox is a versatile package, designed for both simple and complicated graphics applications. Simple procedures allow you to draw

- Points
- Lines
- Rectangles with optional shading
- Ellipses
- Circles

High-level procedures let you create the more complex graphics that are often needed in business and scientific applications:

- Labeled pie charts
- Bar charts with programmable shading
- A variety of curves, using different linestyles and with optional smoothing
- Curve fitting
- Line and solid modeling
- Labeled coordinate axes
- Polygons of any shape, with optional rotation or translation

All your drawings can be displayed either on the full screen, or in windows that you define. You can also draw on a RAM (virtual) screen in memory, without display, and move the resulting images to the displayed screen when desired.

Here are some examples of the kind of drawings you'll soon be able to generate with the Graphix Toolbox.

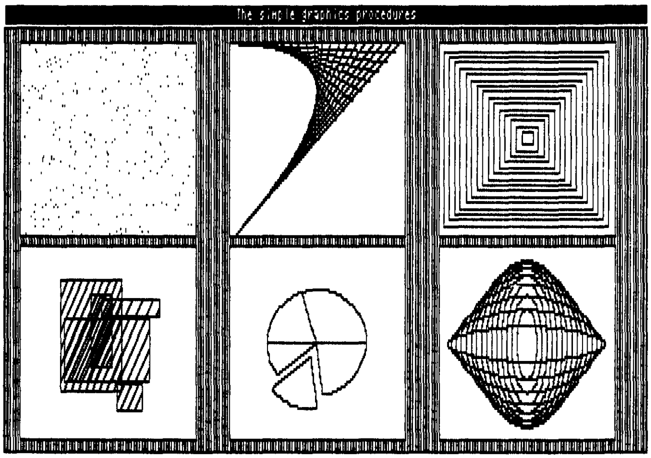


Figure 1 *A Sampler of Drawings Done with the Graphix Toolbox*

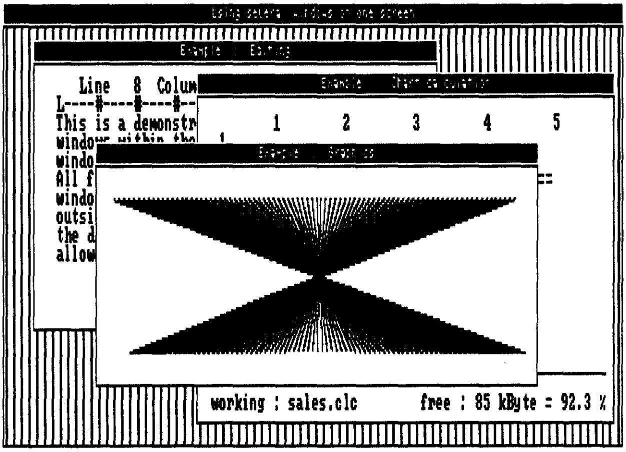


Figure 2 Stacked Windows

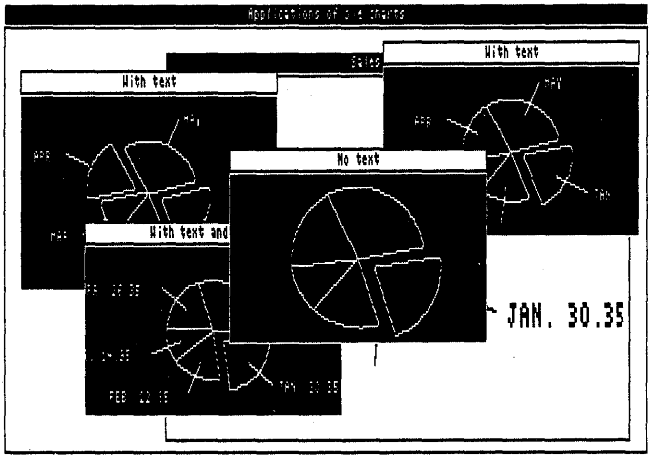


Figure 3 Variations on a Pie Chart

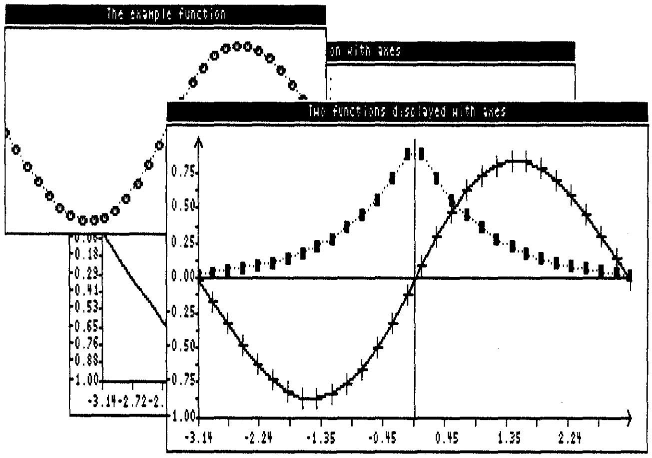


Figure 4 Two Curves Displayed with Coordinate Axes

Structure of This Manual

This manual is divided into five parts:

- Chapter 1 provides an overview of the Turbo Pascal Graphix Toolbox. Basic graphics terms you need to know in order to use the toolbox are defined, and illustrations of some of the things you can draw are given. This chapter also talks about the different hardware configurations that can run the Turbo Pascal Graphix Toolbox.
- Chapter 2 gets you started on using the Turbo Pascal Graphix Toolbox. Turbo Pascal examples for the most commonly used procedures are given, along with the resulting drawings. You'll also see how to define and manipulate windows, and save and print the graphic images you create.
- Chapter 3 is the technical reference part of the manual. All the constants, types, procedures, and functions contained in the Turbo Pascal Graphix Toolbox are described, in alphabetical order, with parameters, function, restrictions, and examples.
- Appendix A explains how to use the Turbo Pascal Graphix Toolbox with different hardware configurations.
- Appendix B provides a glossary of terms used in the manual.

Typography

The body of this manual is printed in normal typeface. Special characters are used for the following special purposes:

`Alternate`	`Alternate characters are used in program examples and procedure and function declarations.`
Italics	*Italics are used to emphasize certain concepts and terminology, such as predefined standard identifiers, parameters, and other syntax elements.*
Boldface	**Boldface type is used to mark reserved words, in the text as well as in program examples.**

Refer to the *Turbo Pascal Reference Manual* for a complete description of the syntax, special characters, and overall appearance of the Turbo Pascal language.

The Distribution Disk

The Turbo Pascal Graphix Toolbox distribution disk contains the following:

- Installation and demonstration files
- Files containing all the procedures and functions
- All the commented program examples used in Chapter 2

The distribution disk is your only source for the Turbo Pascal Graphix Toolbox files. The first thing you should do upon receiving the disk is to complete and mail the License Agreement at the front of this manual. You should then make a copy of the distribution disk. Put the original disk in a safe place, and use only the copy for doing your work. You should never use the distribution disk for your work, since there is a charge for a replacement copy.

After you complete the License agreement and make a backup copy of the disk, read the README.COM file on the disk. Simply type README from the DOS prompt, and the file will appear. It contains a list of files on the disk, as well as any corrections, comments, or updates on the program.

Acknowledgments

In this manual, references are made to several products:

- Flight Simulator is a registered trademark of Sublogic Inc.
- Hercules is a registered trademark of Hercules Computer Technology, Inc.
- IBM is a registered trademark of International Business Machines Inc.
- MS-DOS is a registered trademark of Microsoft Inc.
- Turbo Pascal is a registered trademark of Borland International Inc.

C H A P T E R 1

A Computer Graphics Primer

Before you do any drawing with the Turbo Pascal Graphix Toolbox, you will need to understand the graphics and screen display terms used throughout this manual. Each of these concepts is described here, followed by a list of the Turbo Pascal Graphix procedures and functions that apply to each.

Pixels

The term *pixel* is an acronym for *picture element.* Pixels, in fact, are the basic elements that make up a video display image. The tiny dots that combine to make the text and graphic images you see on your computer monitor are pixels.

The Turbo Pascal Graphix Toolbox allows you to display pixels as black or white with monochrome cards, or in any color supported by a color card.

Screens

A *screen* is the configuration of pixels that make up displayed text or graphic images. Depending on the type of graphics card installed in your system, the screen display will be made up of the following horizontal-by-vertical pixel dimensions:

- IBM CGA 640x200
- Hercules 720x350
- AT&T 6300 640x400
- IBM 3270 720x350
- IBM EGA 640x350

Because the Hercules display is made up of a greater number of pixels, the graphic images created are finer in grain—that is, they are higher in *resolution*. Because of their higher resolution, they also take longer to draw. IBM CGA graphics images are coarser grained, and therefore lower in resolution. The concept of resolution is easy to understand if you think of drawings made with pencils or pens; a drawing done with a fine-point drawing pen will be of a higher resolution, and will take longer to draw than one done with a blunt pencil.

For standard text display—that is, the text normally displayed by your system—a screen can also be thought of as a sequence of 80 vertical character columns that make up the width, and 25 lines of characters that make up the height.

There are two types of screens that you can use for creating images with the Toolbox: the screen displayed on your monitor, and a RAM (virtual) screen in memory. You can draw on either screen, but only the monitor screen is viewable; the RAM screen is invisible. The screen you are currently drawing on is called the *active screen*. RAM screens are useful for storing complicated images that are used often and are time consuming to redraw, or for animation, when it would be distracting to allow the computer to visibly redraw the screen.

The procedures and functions that are used to manipulate screens are

ClearScreen	LoadScreen
CopyScreen	SaveScreen
GetScreen	SelectScreen
InvertScreen	SwapScreen

Characters and Fonts

A *character* is a letter, number, or symbol that is represented on your screen by a rectangular configuration of pixels. A sequence of characters makes up a display of *text.*

There are two styles—or *fonts*—in which text can be displayed with the Turbo Pascal Graphix Toolbox:

- A simple, 4x6-pixel upper- and lower-case font that is used to display window headers, pie chart labels, or any text you wish to display in integer multiples of 4x6 pixels
- A larger, higher quality font (8x8 pixels with an IBM CGA, or the 3270, and the AT&T 6300, and 9x14 pixels with the IBM EGA, or the 3270, and the Hercules monochrome card) that corresponds to the font normally used with the particular graphics card installed in your system

Exactly how the Turbo Pascal Graphix Toolbox utilizes these two fonts will become clear when you read the next section about coordinate systems.

The procedures and functions that affect text are

DC	DrawTextW
DefineHeader	TextDown
DefineTextWindow	TextLeft
DisplayChar	TextRight
DrawAscii	TextUp
DrawText	

Coordinate Systems

A *coordinate system* is a method used to identify a location according to its position relative to horizontal and vertical axes. In mathematics, usually, and in Turbo Pascal Graphix Toolbox programming in particular, the horizontal axis is labeled *X*, and the vertical axis *Y*. The exact location of, for example, a point, is determined by the *X* and *Y* coordinates of that point – that is, its distance from the *X* and *Y* zero axes.

Coordinate systems are extremely important in graphics programming, since all screen positions for text and graphics must be specified using *X* and *Y* coordinates. There are two types of coordinate systems that you can choose when working with the Turbo Pascal Graphix Toolbox: absolute screen and world coordinate systems.

Absolute Screen Coordinate System

The *absolute screen coordinate system* refers to the entire monitor screen, and the actual character and pixel screen positions, for plotting text and graphics; coordinates [0,0] are in the upper left corner of the screen, with the *X* coordinates increasing to the right, and the *Y* coordinates increasing downward. As mentioned earlier, the screen can be regarded either as a configuration of pixels or as a series of 25 lines by 80 columns.

Text is handled in two ways. The simple, 4x6-pixel font used for window headers and footers can be plotted anywhere on the screen, and can be scaled to be any size that is an integer multiple of 4x6 pixels (for example, 8x12). The higher quality font is plotted according to 80x25 text column and line coordinates.

World Coordinate System

For most graphics, the absolute screen coordinate system will not easily translate to the application's numeric values. A *world coordinate system* is an arbitrary coordinate system that you specify to accommodate your particular application. The numbers you use in your world coordinate system can be (and usually are) completely unrelated to pixel coordinates. In Turbo Pascal Graphix Toolbox language, this is called *defining a world.*

A world coordinate system is usually used to scale images so that they fit correctly into the windows you have defined. After you define the world for a given window, any images you subsequently draw will be automatically, proportionately scaled to fit the window.

The procedures and functions that affect worlds are

DefineWorld	ResetWorlds
FindWorld	SelectWorld

Windows

A *window* is any area of the screen that you define as the drawing area. Several windows, containing different drawings and text, can be displayed simultaneously on the screen. Each window can be moved independently of the other windows, placed on top of other windows, and stored to, recalled from, or erased from memory. Windows can be stored and loaded individually or in groups to and from disk. Several windows can be stored in RAM, and quickly copied to and from the active

screen. You can draw borders, incorporate high-quality text, and label your windows with headers or footers. The window you are currently drawing in is called the *active window*.

A window can be specified to be almost any size, from the whole screen to 1 vertical pixel by 8 horizontal pixels. You define a window area by specifying the *X* and *Y* coordinates of its upper left and lower right corners, with *Y* coordinates measured in 1-pixel units and *X* coordinates measured in 8-pixel units. These coordinates are called *window definition coordinates*. In window definition coordinates, the point [0,0] refers to the upper left corner of the screen.

Once you're working within a window, you can redefine its world coordinate system, thereby allowing multiple images to be displayed within one window, each with its own coordinate system. Coordinate axes, along with lettering, can be easily added to any drawing.

A special RAM memory area, the *window stack*, is set aside for temporary storage of windows. The stack comes in handy when you have several windows that you want to keep but don't want to display all at the same time. The stack is also used for storing windows that would otherwise be erased when another window is moved over them on the screen.

The procedures and functions that affect windows are

ClearWindowStack	RedefineWindow	SetClippingOff
Clip	RemoveHeader	SetHeaderOn
Clipping	ResetWindows	SetHeaderOff
CopyWindow	ResetWindowStack	SetHeaderToTop
DefineHeader	RestoreWindow	SetHeaderToBottom
DefineWindow	SaveWindow	SetWindowModeOff
DefineWorld	SaveWindowStack	SetWindowModeOn
DrawBorder	SelectWindow	StoreWindow
GetWindow	SelectWorld	WindowSize
InvertWindow	SetBackground	WindowX
LoadWindow	SetBackground8	WindowY
LoadWindowStack	SetClippingOn	

Clipping

The Turbo Pascal Graphix Toolbox allows you to "clip" images at window boundaries if you wish. This feature accomplishes several purposes:

- It relieves you from having to be exact when you're drawing in a window. The Toolbox does the nitty-gritty of keeping your work within window boundaries.

- It lets you "zoom in" on some aspect of a drawing. For example, let's say you've defined your world coordinate system for a window. Once you're working in the window, you can redefine the world. When the image is drawn, the Turbo Pascal Graphix program will "zoom in" and "clip" any part of your drawing that falls outside the window with the new coordinate system.
- It protects program memory. Drawings that stray outside screen boundaries can encroach on other parts of memory, including parts of your application program.

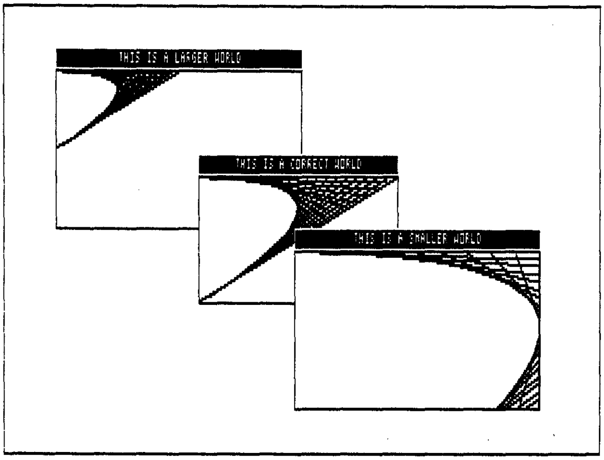


Figure 1-1 *The Clipping Option Used To "Zoom In" on a Drawing*

There are times when you'll choose not to clip drawings, too. For instance, you may develop a program using the clipping option, but once the program is debugged, and you know your drawings are within bounds, you can turn clipping off. This speeds up the drawing process considerably. Or, if you're working strictly with absolute coordinates, you don't need to worry about drawing outside screen boundaries.

How to Use the Turbo Pascal Graphix Toolbox With Your Hardware

There are a few differences between the computer systems and graphics cards that can run the Toolbox. In some cases, these differences require your special consideration when creating Toolbox-based programs.

There are two hardware considerations to take into account if you are using the IBM version of the Turbo Pascal Graphix Toolbox: IBM compatibility, and graphics

cards. The information below will tell you briefly what you need to know about your particular system; more technical details about certain hardware configurations can be found in Appendix A.

The IBM PC and True Compatibles

The Turbo Pascal Graphix Toolbox runs on any IBM PC, PC Jr., and compatible computer. But what exactly is a *true* IBM-compatible computer? There are many computers on the market today that are billed as IBM-compatible, and to some extent they are. However, when considering whether a computer is IBM compatible, it is important to look at the specific application you are using the computer for. In the case of the Turbo Pascal Graphix Toolbox, you must consider whether the graphics displayed by your computer will be true to your program design.

A potential problem with some IBM compatibles is that their screen display is of a higher resolution than the IBM screen. The Corona PC is a good example. Although the Corona's higher resolution display can make for very high-quality text and graphics, graphic images created with the Turbo Pascal Graphix Toolbox will not display true-to-form on the Corona screen; because of the Corona's higher resolution, the drawing will appear to be compressed vertically.

A good test for whether your IBM-compatible computer will run the Toolbox is to test the Flight Simulator program (written for the IBM PC) on your system. If your computer can run Flight Simulator, it's a good bet it will also run the Toolbox without problems.

Compatibility is also a consideration when your program will be running on more than one computer system. Some distortion of screen images may result when a program designed on a computer with an IBM card is run on a computer with a Hercules card. See Appendix A for information about how to cope with those kinds of problems.

Following is a list of computers and graphics cards that are sure to run the Turbo Pascal Graphix Toolbox. Next to the name of the product, the Graphix Toolbox graphics driver that runs with that product is given in parentheses. If your computer or graphics card is not on this list, give a call to Borland's technical support staff; they'll be able to tell you whether your computer will run the Graphix Toolbox.

AT&T PC 6300 (CGA)
AT&T (640x400) (ATT)
Columbia MBC, VP (CGA)
Compaq Portable and DeskPro (CGA)
Comway Comgraphics card (HGC)
Comway Comtronics (CGA)

Comway Comcolor (CGA)
Heath/Zenith Z150 series (CGA)
Hercules color card (CGA)
Hercules monochrome card (HGC)
IBM Color/Graphics adapter (CGA)
IBM Enhanced Graphics adapter or EGA-compatible cards (EGA)
IBM PCjr (CGA)
IBM 3270 (3270)
Leading Edge PC (CGA)
MA Systems PC Peacock (CGA)
Panasonic SR Partner (CGA)
Paradise/USI MultiDisplay (CGA)
Paradise Modular Graphics Card (CGA)
Profit Systems Multigraph (CGA)
QuadRAM QuadColor I,II (CGA)
Seequa Chameleon line (CGA)
STB Graphics Plus II (CGA)
Tandy 1000 (CGA)
Tava (CGA)
Tecmar Graphics Master (CGA)
TeleVideo PC (CGA)
Tseng Laboratories UltraPAK (HGC)
Vutek Color Plus (CGA)

IBM Color Graphics Card

If you have an IBM graphics card installed in your computer, your screen display is 640 pixels wide by 200 pixels tall. The *SetBackgroundColor* and *SetForegroundColor* procedures are used to determine background and display image colors. You can also use the *SetColorWhite* and *SetColorBlack* procedures to reverse the background and foreground colors.

Hercules Monochrome Graphics Card

The Hercules graphics card produces a higher resolution display: 720 pixels wide by 350 pixels tall. The background of the display will be black, and the displayed images will be in the color produced by your monochrome monitor.

There are some important considerations to keep in mind when you decide to run your programs developed with a Hercules card on other systems. These and other potential problems are discussed in Appendix A.

IBM Enhanced Graphics Adapter

With the IBM Enhanced Graphics Adapter you get a high resolution display of 640 pixels wide by 350 pixels tall. The *SetBackgroundColor* and *SetForegroundColor* procedures may be used to determine background and display image colors; and the *SetColorWhite* and *SetColorBlack* procedures to reverse the background and foreground colors.

IBM 3270 PC

The IBM 3270 PC's graphics card produces a high resolution display: 720 pixels wide by 350 pixels tall. The *SetBackgroundColor, SetForegroundColor, SetColorWhite* and *SetColorBlack* procedures may be used to determine the background and foreground colors.

The AT&T 6300

The AT&T 6300's graphics card gives you a high resolution display of 640 pixels wide by 400 pixels tall.

You can use the same procedures to determine colors and color reversals.

C H A P T E R 2

Getting Started

Ready to start drawing? This tutorial chapter takes you on a step-by-step tour of the Turbo Pascal Graphix Toolbox, using commented program examples for both basic and sophisticated graphics routines. The examples build on each other, so if you read the chapter through in order, by the end you should be ready to incorporate the Turbo Pascal Graphix routines you need into any graphics application program.

This chapter is designed as a basic tutorial. Technical details about the Turbo Pascal Graphix procedures used in this chapter can be found in Chapter 3. Basic graphics concepts and terminology used in this chapter are explained in Chapter 1 and Appendix B.

Linking Turbo Pascal Graphix Routines into Your Program

To use the Turbo Pascal Graphix Toolbox, you must first incorporate the two basic Toolbox units into your program with the Turbo Pascal **uses** clause. The **uses** clause specifies which precompiled units will be linked into your application program. It begins with the keyword **uses** followed by a list of unit names separated by commas. You should enter the **uses** clause at the top of your program, just below the program declaration and above your constant, type, and variable declarations.

Every Turbo Pascal Graphix program must use the following system and toolbox units in the order given:

```
uses
  Dos, Crt, GDriver, Printer, GKernel;
```

The units *Dos*, *Crt*, and *Printer* are Turbo Pascal standard units. The units *GDriver* and *GKernel* are Graphix Toolbox units.

To install the toolbox for your particular graphics card, you must copy the device driver written for your hardware (supplied on the distribution disk) onto the GDRIVER.PAS file. This is done by invoking the Turbo Pascal Graphix batch program *Tginst*. The batch program *Tginst* takes a command line argument that specifies which graphics card you are installing the toolbox for. The following list shows the command line argument you should use for particular graphics cards:

CGA – for the IBM Color graphics adapter.

EGA – for the IBM Enhanced graphics adapter.

3270 – for the IBM 3270 PC.

HGC – for the Hercules monochrome card.

ATT – for the AT&T 6300 PC.

For example, to install the toolbox for use on a Hercules monochrome card you would type the following at the DOS prompt:

```
tginst HGC
```

Next, before calling the Turbo Pascal Graphix routines you need for your particular application, you must initialize the graphics system by calling the *InitGraphic* procedure. At the end of your program, you must call *LeaveGraphic* to return your system to text mode. See Chapter 3 for detailed information about these procedures.

All of the example programs in this chapter are included on the Turbo Pascal Graphix Toolbox distribution disk, so you can try out the examples and experiment with the calling parameters in the various procedures. Each example program is listed under a file name of the form *filename*.PAS.

Every program example consists of five basic steps:

- Include at least the two core Turbo Pascal Graphix units
- Call *InitGraphic* to enter graphics mode
- Call *DrawBorder* to draw a border around the drawing area (optional)
- Draw your images or text
- Include a wait loop so you can view the display (optional)
- Call *LeaveGraphic* to return to text mode

8087 or Non-8087 Installation

The file FLOAT.INC on the distribution diskette contains the definition for the user-defined type Float that is used thoughout the toolbox whenever a real number is needed. FLOAT.INC is included in the file GDRIVER.PAS and must be present if you want to rebuild this unit. The definition of the type Float is equivalent to using either the standard Turbo Pascal type real or type double. The reason that we redefine all real numbers to Float in the toolbox is to aid the user in being able to select which real number type they want to use by simply changing the value of the $N compiler directive in the file FLOAT.INC to select between using the double precision 8087 real {$N+} or the standard non-8087 real number {$N-}. The advantage to using the double precision real number is an increase in speed of all real number calculations due to the fact that the 8087 coprocessor is required and utilized for all floating point operations. The advantage to using the standard Turbo Pascal six byte real number is that an 8087 chip isn't required and you save two bytes of data space for each real number that is declared. To install the toolbox for non-8087 support simply change the $N compiler directive in the file FLOAT.INC to {$N-} and then rebuild all of the toolbox units with the build option in the compiler. To install the toolbox to utilize the 8087 math coprocessor chip, you need to set the $N compiler directive to {$N+} in the file FLOAT.INC and then rebuild all of the toolbox units with the **Build** option in the compiler. All real variables should be declared as type Float in programs that use the toolbox routines.

Drawing Points

You can use the Turbo Pascal Graphix *DrawPoint* procedure to draw points using either absolute screen or world coordinates. (See Chapter 1 for a definition of coordinate systems.) The next two sections show you how to draw points using the screen coordinate system, while the section following explains how points are drawn in world coordinates. You should read this section even if you aren't interested in drawing points, because the rest of the examples in this chapter utilize world coordinate systems; it is important that you understand the point-drawing examples in order to see the difference between screen and world coordinate systems.

Drawing a Single Point

Writing a program that draws a single point is the simplest thing you can do with the Turbo Pascal Graphix Toolbox. Below is a Turbo Pascal program (DRWPNT.PAS on the distribution disk) that draws and displays a single point.

```
program ExamplePoint;
uses
  Dos, Crt, GDriver, Printer, GKernel;

begin
  InitGraphic;              { Initialize (init) the graphics system }
  DrawBorder;               { Draw a border around the default window }
  DrawPoint(100, 100);      { Plot a pixel at (100, 100) }
  repeat until KeyPressed;  { Wait until a key is pressed }
  LeaveGraphic;             { Leave the graphics system }
end. { ExamplePoint }
```

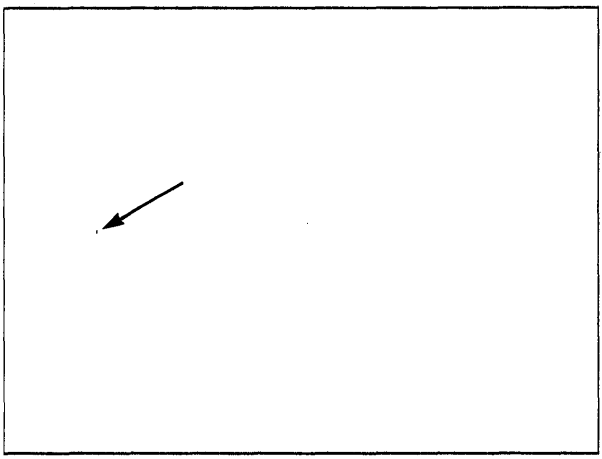


Figure 2-1 A Single Point (DRWPNT.PAS Example)

Drawing a Cluster of Points

The following program (DRWPNTS.PAS on the distribution disk) draws 1000 points, displayed randomly on the screen. For this example, let's assume you have an IBM graphics card installed in your system.

```
program DrawPoints;

uses
  Dos, Crt, GDriver, Printer, GKernel;

var
  I : integer;

begin
  InitGraphic;               { Init the system and screen }
  DrawBorder;
  for I := 1 to 1000 do         { Draw 1000 random points on IBM screen }
    DrawPoint(Random(XScreenMaxGlb), Random(YMaxGlb)); { Plot random pixels on
                                                         the screen }
  repeat until KeyPressed;              { Wait until a key is pressed }
  LeaveGraphic;
end. { DrawPoints }
```

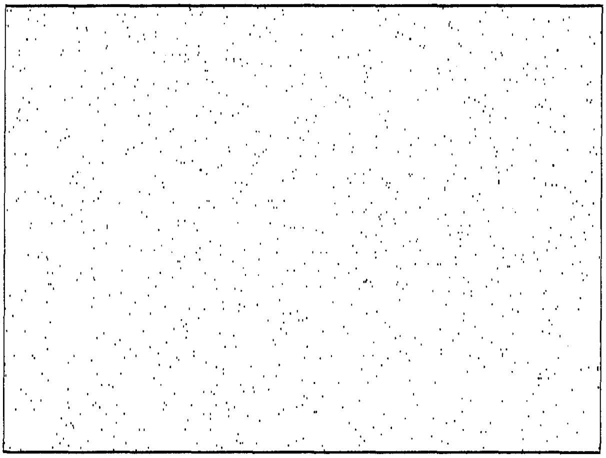


Figure 2-2 A Cluster of Points (DRWPNTS.PAS Example)

To allow you to run your program on systems with different graphics cards, you can write this program so that it uses a world coordinate system instead of the absolute screen coordinate system, as described next.

Drawing Points Using a World Coordinate System

A *world coordinate system* lets you define the addresssing dimensions of your drawing area, independently of the screen type and size. Once you have defined your world, the Turbo Pascal Graphix program will scale the drawing to fit the screen or window you are using.

The following program (WDRWPNTS.PAS on the distribution disk) is identical to the one in the previous section, but uses a world coordinate system instead of the absolute screen coordinate system.

```
program WorldDrawPoints;

uses
  Dos, Crt, GDriver, Printer, GKernel;

var
  I : integer;

begin
  InitGraphic;                    { Init the system and screen }
  DrawBorder;

  DefineWorld(1,0,0,1000,1000);   { Define a world for drawing }
  SelectWorld(1);                 { Select it }
  SelectWindow(1);

  for I := 1 to 1000 do           { Draw 1000 random points on world }
    DrawPoint(Random(1000), Random(1000));

  repeat until KeyPressed;        { Wait until a key is pressed }
  LeaveGraphic;
end. { WorldDrawPoints }
```

Erasing a Point

To erase a point, change the drawing color to black and then draw the point, as follows:

```
SetColorBlack;
DrawPoint(x,y);
```

Summary of Point Routines

- *DrawPoint* draws a point in world or screen coordinates.
- *DP* draws a point in absolute screen coordinates only.
- *PD* returns TRUE if a point is drawn in specified screen coordinates.
- *PointDrawn* returns TRUE if a point is drawn at specified world coordinates.

Drawing Lines

The *DrawLine* procedure allows you to draw and display lines in the current line style (selected by the *SetLineStyle* procedure). The coordinates for lines drawn in the following program examples are all calculated using world coordinate systems.

Drawing a Single Line

The following program (DRWLIN.PAS on the distribution disk) draws a line from the upper left to the lower right corner of the screen. Endpoint coordinates are passed to the procedure as the *X* and *Y* coordinates of the first endpoint, followed by the *X* and *Y* coordinates of the second endpoint.

```
program DrawLine;

uses
  Dos, Crt, GDriver, Printer, GKernel;

begin
  InitGraphic;                    { Init the graphics system }
  DrawBorder;
  DefineWorld(1,0,0,1000,1000); { Define the world to draw in }
  SelectWorld(1);                 { Select the world }
  SelectWindow(1);                { Select the window }

  DrawLine(0,0,1000,1000);        { Draw the line }

  repeat until KeyPressed;        { Wait until a key is pressed }
  LeaveGraphic;                   { Leave the graphics system }
end. { DrawLine }
```

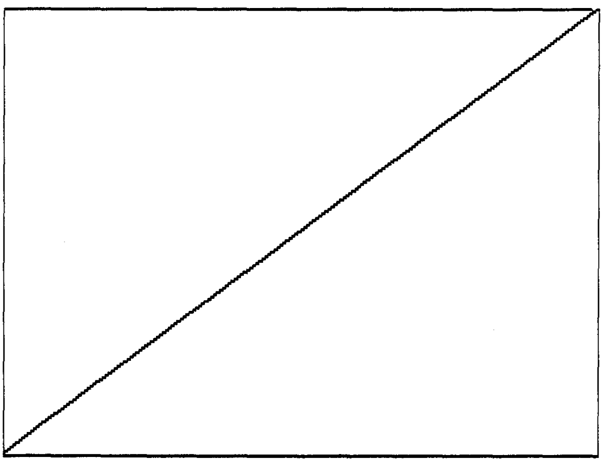


Figure 2-3 *A Line (DRWLIN.PAS Example)*

Drawing a "Walking Line"

An intriguing variation on the *DrawLine* procedure is the "walking line." A walking line program generates, by increments, a series of endpoint coordinates, thereby creating a "walking line." By changing the formula used to generate the endpoint coordinates, a variety of shapes can be drawn. In the example that follows (DRWLINS.PAS on the distribution disk), the first endpoint moves uniformly across the top of the screen from left to right, while the other endpoint moves incrementally and diagonally from the upper right to the lower left corner of the screen.

```
program DrawLines;

uses
  Dos, Crt, GDriver, Printer, GKernel;

var
  I : integer;

begin
  InitGraphic;                    { Init the system and screen }

  DefineWorld(1,0,0,1000,1000);   { Define a world for drawing }
  SelectWorld(1);                 { Select it }
  SelectWindow(1);

  SetBackground(0);               { Set the background color to black }
  DrawBorder;

  for I := 1 to 20 do             { Draw 20 lines }
    DrawLine(I*50, 0, 1000-I*50, I*50);

  repeat until KeyPressed;        { Wait until a key is pressed }
  LeaveGraphic;
end. { DrawLines }
```

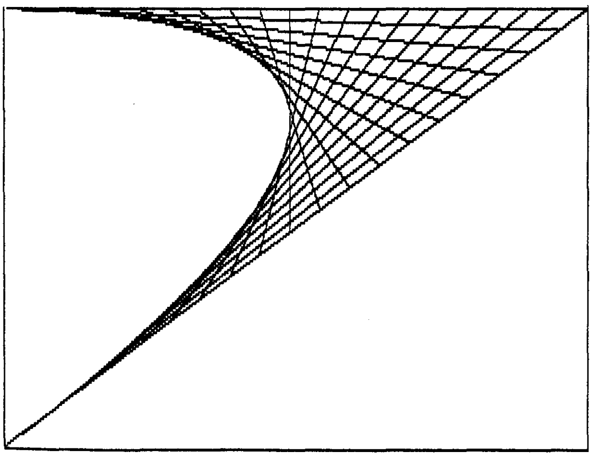


Figure 2-4 *A Walking Line (DRWLINS.PAS Example)*

Summary of Line-Drawing Routines

- *Clip* clips a line at active window boundaries.
- *DrawLine* draws a line using world or screen coordinates.
- *DrawLineClipped* clips a line at screen boundaries.
- *DrawStraight* draws a horizontal line.
- *SetLinestyle* selects one of five linestyles for drawing lines.
- *GetLineStyle* returns the current linestyle.

Drawing Squares

The *DrawSquare* procedure draws rectangles in the current line style (selected by the *SetLineStyle* procedure). A rectangle is defined by the coordinates of the points at its upper left and lower right corners. A Boolean value, *Fill* allows you to fill the rectangle with the current drawing color (determined by the *SetForegroundColor* procedure). The following program (DRWSQ.PAS on the distribution disk) draws a series of consecutively larger squares around the center of the screen, with no fill. Another example program not illustrated here (DRWHCH.PAS on the distribution disk) draws hatched squares.

```
program DrawSquares;

uses
  Dos, Crt, GDriver, Printer, GKernel;

var
  I : integer;

begin
  InitGraphic;                    { Init the system and screen }

  DefineWorld(1,0,0,1000,1000);   { Define a world for drawing }
  SelectWorld(1);                 { Select it }
  SelectWindow(1);
  DrawBorder;

  for I := 1 to 20 do             { Draw 20 squares }
    DrawSquare(500-I*25, 500-I*25, 500+I*25, 500+I*25, false);

  repeat until KeyPressed;        { Wait until a key is pressed }
  LeaveGraphic;
end. { DrawSquares }
```

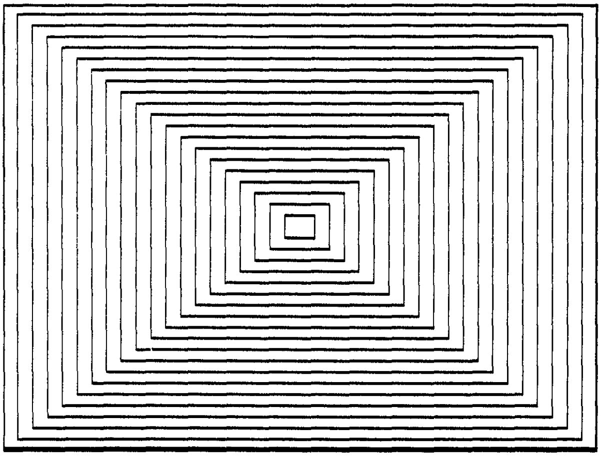


Figure 2-5 Squares (DRWSQ.PAS Example)

Summary of Square-Drawing Routines

- *DrawSquare* draws a square using world coordinates.
- *DrawSquareC* draws a square using screen. coordinates, but clipped at the boundaries of the active window.
- *SetForegroundColor* chooses the current drawing color.
- *SetLinestyle* chooses the line style.

Drawing Circles

Because different graphics cards produce screen displays with different vertical-by-horizontal dimensions, and because different monitors have different screen proportions, a correctly-proportioned circle drawn on one screen may look distorted on another screen. To adjust for differences in screen proportions, Turbo Graphix routines that deal with circles and ellipses—*DrawCircle, DrawCircleSegment, DrawCartPie* and *DrawPolarPie*—utilize the concept of the *aspect ratio.*

An aspect ratio is defined as the height-to-width ratio of a circle or ellipse. Turbo Pascal Graphix circle routines allow you to vary the aspect ratio's vertical dimension by calling the *SetAspect* procedure. In addition, a global constant, *AspectFactor,* sets the system-dependent aspect ratio, so that an aspect ratio of 1.0 produces a true circle on a particular hardware screen.

The following program (DRWCIR.PAS on the distribution disk) draws a series of circles, and varies both their radii and aspect ratios. The parameters passed to the *DrawCircle* procedure specify the *X* and *Y* world coordinates of the center of the circle; the radius corresponds to the *X*(horizontal) dimension of the circle.

```
program DrawCirc;

uses
  Dos, Crt, GDriver, Printer, GKernel;

var
  I : integer;
  AspectLoc, Rad : Float;

begin
  InitGraphic;                    { Init the system and screen }

  DefineWorld(1,0,0,1000,1000);   { Define a world for drawing }
  SelectWorld(1);                 { Select it }
  SelectWindow(1);
  DrawBorder;

  Rad := 1.5;                     { Set initial radius }
  AspectLoc := GetAspect;         { Save default aspect ratio }
  SetAspect(0.2);                 { Init aspect ratio for this routine }

  for I := 1 to 15 do             { Draw circles }
  begin
    DrawCircle(500, 500, Rad);
    SetAspect(0.2 + I/10);
    Rad := Rad - 0.05;
  end;

  SetAspect(AspectLoc);           { Restore previous aspect ratio }

  repeat until KeyPressed;        { Wait until a key is pressed }
  LeaveGraphic;
end. { DrawCirc }
```

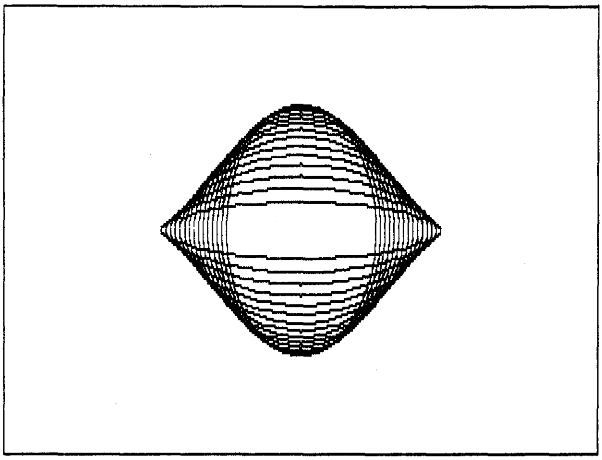


Figure 2-6 *Circles (DRWCIR.PAS Example)*

Summary of Related Routines

- *DrawCircle* draws a circle or ellipse using world or screen coordinates.
- *DrawCircleDirect* draws a circle or ellipse using screen coordinates.
- *DrawCircleSegment* draws an arc of a circle.
- *DrawPie* draws a pie chart.
- *GetAspect* returns the current aspect ratio.
- *SetAspect* determines the aspect ratio for a circle.

Text

As explained in Chapter 1, the Turbo Pascal Graphix Toolbox supports both a 4x6-pixel text and a machine-dependent text. The size of machine-dependent characters is 8x8 pixels for IBM CGA and AT&T, and 9x14 pixels for Hercules, IBM EGA, and IBM 3270.

Displaying Machine-Dependent Text

The text routines used by the Turbo Pascal Graphix Toolbox are very similar to those used by Turbo Pascal; the screen is defined as 25 lines by 80 columns (characters), and the Turbo Pascal procedures *GotoXY*, *Write* and *WriteLN* are supported by the Graphix Toolbox. However, there are a few considerations specific to the Turbo Pascal Graphix text mode concerning the alignment of text with drawings, and within windows. Since the size of the text font varies with the graphics card installed, some adjustments must be made when attempting to align text with drawings. In particular, Hercules text, which is defined on a 9-pixel horizontal boundary, must be adjusted for the 8-pixel window boundary. See Appendix A for technical information on text fitting.

The following program (DRWSTXT.PAS on the distribution disk) places the start of a text string at the center of the screen, demonstrates the automatic new-line performed by *Writeln*, and places the text within a filled box whose dimensions are determined according to the world coordinate system. The coordinates for the points at the corners of the box are computed from the character positions of the text.

```
program DrawStandardText;

uses
  Dos, Crt, GDriver, Printer, GKernel;

const
  MaxWorldX : Float = 1000.0;
  MaxWorldY : Float = 1000.0;

var
  I : integer;
  CharHeight, CharWidth : Float;

begin
  InitGraphic;                       { Init the graphics system }

  DefineWorld(1, 0, 0, MaxWorldX, MaxWorldX);    { Define the world to draw in }
  SelectWorld(1);                    { Select the world and window }
  SelectWindow(1);
  DrawBorder;

  GotoXY(39, 12);                    { Goto the center of the text screen }
  Writeln('* <- This should be at the center '); { Write two lines of text }
  Write('This should be on the next line');

  CharWidth := MaxWorldX / 80;       { Compute a character's width }
  CharHeight := MaxWorldY / 25;      { Compute a character's height }

  DrawSquare(9*CharWidth, 7*CharHeight,          { Draw box at text loc [10,8] }
          (22*CharWidth)+2, (8*CharHeight)+2, true);

  GotoXY(10, 8);                     { Position cursor }

  Write('Text in a box');            { Write text in it }

  repeat until KeyPressed;           { Wait until a key is pressed }
  LeaveGraphic;                      { Leave the graphics system }
end. { DrawStandardText }
```

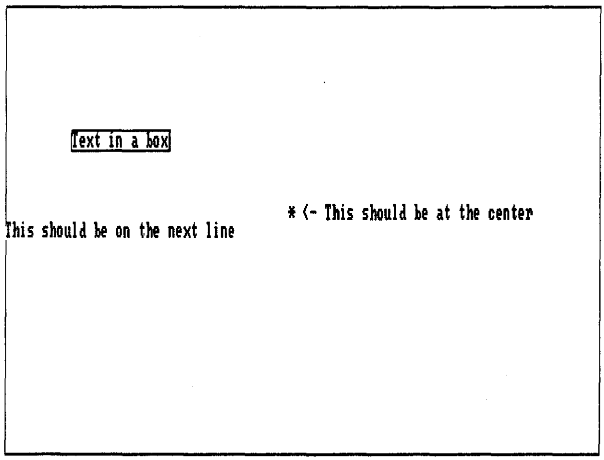


Figure 2-7 Machine-Dependent Text (DRWSTXT.PAS Example)

The 4x6 pixel character set is used for window headers, and for applications that require text that is smaller or larger than the machine-dependent text. Unlike the machine-dependent text, the 4x6 pixel characters can be placed at any screen location. The *Scale* parameter passed to the *DrawText* procedure specifies the size of the characters (in integer multiples of 4x6 pixels); the larger the value of *Scale*, the larger the character.

Since a character in the 4x6 pixel font is made up of only a few pixels, this text is of a coarser quality than the machine-dependent text, even when they are scaled to the same size.

The following example (DRWATXT.PAS on the distribution disk) uses the *DrawText* procedure to display upper-case characters, in different positions and sizes, in the center of the screen. The complete character set is then displayed at the upper left corner of the screen, scaled to its smallest size.

```
program DrawAlternateText;

uses
  Dos, Crt, GDriver, Printer, GKernel;

const
  MaxWorldX  : Float = 1000.0;
  MaxWorldY  : Float = 1000.0;
  CharArray1 : array [0..25] of char =       { Define an array of characters }
                      ('A','B','C','D','E','F','G','H','I','J','K','L','M',
                       'N','O','P','Q','R','S','T','U','V','W','X','Y','Z');

var
  I : integer;
  CharHeight, CharWidth : Float;

begin
  InitGraphic;                               { Init the graphics system }

  DefineWorld(1, 0, 0, MaxWorldX, MaxWorldY);  { Define the world to draw in }
  SelectWorld(1);
  SelectWindow(1);
  DrawBorder;

  for I := 1 to 50 do          { Print Random characters in center of screen }
    DrawTextW(Random(600) + 200, Random(600) + 200,
              Random(5), CharArray1[Random(26)]);

  DrawTextW(15, 50, 1, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'); { Type chars in corner }
  DrawTextW(15, 100, 1, 'abcdefghijklmnopqrstuvwxyz');
  DrawTextW(15, 150, 1, '1234567890-=\~~!@#$%^&*()_');
  DrawTextW(15, 200, 1, '[]{}:";,.<>/?+|');

  repeat until KeyPressed;                    { Wait until a key is pressed }
  LeaveGraphic;                                 { Leave the graphics system }
end. { DrawAlternateText }
```

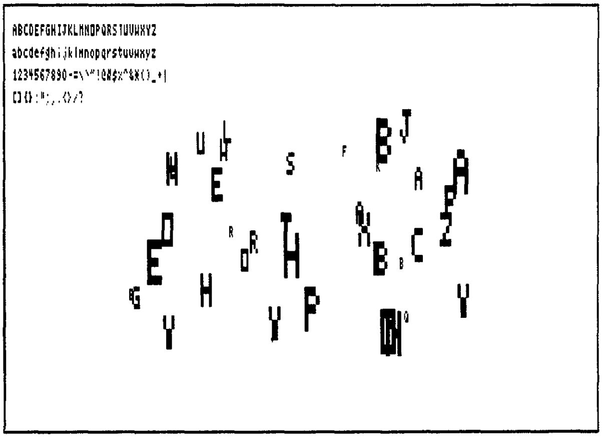


Figure 2-8 *4x6 Pixel Text (DRWATXT.PAS Example)*

Summary of Text-Drawing Routines

For machine-dependent text:

- *DC* draws a character at the specified text coordinates.
- *DefineTextWindow* uses specified text coordinates to define a window.
- *DisplayChar* draws a character at the specified text coordinates.
- *TextDown, TextLeft, TextRight, TextUp* adjust space between window boundaries and text (text fitting).

For 4x6 pixel text:

- *DefineHeader* defines a window header.
- *DrawAscii* draws a character at the specified screen coordinates.
- *DrawText* draws a character string at the specified screen coordinates.
- *DrawTextW* draws a character string at the specified world coordinates.

Windows

This section tells you how to create and manipulate on-screen windows. The use of windows allows greater flexibility in graphics applications, since you can display several different drawings on the screen at once, using different world coordinate systems; and you are not limited to the pixel dimensions of the window.

Defining a Window

When the Turbo Pascal Graphix Toolbox is initialized with the *InitGraphic* procedure, the entire screen is, in effect, defined as a window whose world coordinates correspond to the pixel dimensions of the screen. However, you can redefine any region of the screen as a window, from an 8x1 pixel (horizontal by vertical) box to the entire screen.

Once defined, a window acts more or less independently of other windows and even the screen. Windows can be small or large, moved around, drawn on with reference to their own coordinate systems and boundaries, and individually removed, stored, and retrieved.

Generally, you will want to define a new world coordinate system for every window you define; otherwise, any drawing you do in a window will take place as if the screen coordinate system were mapped to that window. All drawing routines—except routines internal to the graphics system, routines for machine-dependent text positioning such as *GotoXY*, and window positioning routines—can use world coordinate systems.

To associate a world with a window, you must always call *SelectWorld* before *SelectWindow*. If a new window is subsequently selected, the current world is retained. Thus, to draw alternately in two windows with different worlds, *SelectWorld* must be called before each *SelectWindow*:

```
repeat
  SelectWorld(1);
  SelectWindow(1);
    { Insert code to draw something in window 1
      using world coordinate system 1 }
  SelectWorld(4);
  SelectWindow(2);
    { Insert code to draw something in window 2
      using world coordinate system 4 }
until KeyPressed;
```

Besides simply defining the dimensions of your window, you can label it with a header or footer, fill it in with a color or background pattern, or draw a border around it in any line style. When a new window is defined or an existing window is redefined, the header associated with that window number is destroyed. This means that *DefineWindow* must be called before *DefineHeader*.

To change the dimensions of an existing window, without changing its header, use the *RedefineWindow* procedure.

The following example (SIMPWIND.PAS) shows you how to define a window with a border and a header.

```
program SimpleWindow;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow;

begin
  InitGraphic;                    { Init the graphics system }

  DrawBorder;                     { Draw a border around the drawing }
                                  { area of the primary window }

  DefineWindow(1, 10, 20, XMaxGlb-10, YMaxGlb-20);
                                  { Define a window 80 pixels in from }
                                  { the left and right edges, and 20 }
                                  { from the top and bottom edges }

  DefineHeader(1,'THIS IS AN EXAMPLE WINDOW'); { Give it a header }
  SetHeaderOn;

  DefineWorld(1,0,0,1000,1000);  { Give it a world coordinate system }

  SelectWindow(1);                { Select the window }

  SelectWorld(1); 1st             { Select the world }

  SetBackground(85);              { Give it a grey background }

  DrawBorder;                     { Draw the border }

  repeat until KeyPressed;        { Wait until a key is pressed }

  LeaveGraphic;                   { Leave the graphics system }

end. { SimpleWindow }
```

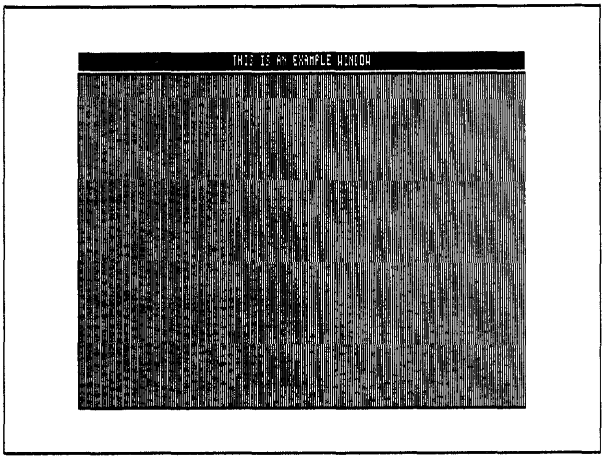


Figure 2-9 *A Window (SIMPWIND.PAS Example)*

Displaying a Drawing in a Window

Suppose you want to display the "walking line" example in a window. You can display the example using a world coordinate system, and in any position on the screen by following these steps:

- Define the window
- Define the world coordinate system for the window
- Select the world coordinate system
- Select the window for drawing
- Draw a border (optional)
- Display the walking lines

The following example (MULTWIND.PAS) displays the walking line example in three different windows, each with its own coordinate system, with the drawings clipped at window boundaries.

```
program MultipleWindows;

uses
  Dos, Crt, GDriver, Printer, GKernel;

var
  I : integer;

procedure DrawLines;

var
  I : integer;
begin
  for I := 1 to 20 do
    DrawLine(I * 50, 0, 1000 - I * 50, I * 50);
end; { DrawLines }

begin
  InitGraphic;              { Init the graphics system }

  DrawBorder;               { Draw a border around the drawing }
                            { area of the primary window }
                            { (the dimensions of the primary window }
                            { default to the screen dimensions) }

  DefineWindow(1, Trunc(XMaxGlb / 10), Trunc(YMaxGlb / 10),
                  Trunc(XMaxGlb / 2), Trunc(YMaxGlb / 2));
                            { Define a window one tenth of the way }
                            { in from the left and top edges, and half }
                            { way down from the right and bottom edges }

  DefineHeader(1, 'THIS IS A LARGER WORLD'); { Give it a header }

  DefineWorld(1, 0, 0, 2000, 2000); { Give it a larger world coordinate system }
```

```
    DefineWindow(2, Trunc(XMaxGlb / 3), Trunc(YMaxGlb / 3),
                    Trunc((XMaxGlb * 2) / 3), Trunc((YMaxGlb * 2) / 3) );
                               { Define a window one third of the way }
                               { in from the left and top edges, and }
                               { from the right and bottom edges }

    DefineHeader(2, 'THIS IS A CORRECT WORLD'); { Give it a header }

    DefineWorld(2, 0, 0, 1000, 1000); { Give it a correct world }
                                      { coordinate system }

    DefineWindow(3, Trunc(XMaxGlb / 2), Trunc(YMaxGlb / 2),
                    Trunc((XMaxGlb * 9) / 10), Trunc((YMaxGlb * 9) / 10));
                               { Define a window one half of the way }
                               { in from the left and top edges, and half }
                               { way down from the right and bottom edges }

    DefineHeader(3, 'THIS IS A SMALLER WORLD'); { Give it a header }

    DefineWorld(3, 0, 0, 500, 500); { Give it a smaller world coordinate system }

    for I := 1 to 3 do
    begin
      SelectWindow(I);         { Select window }
      SetHeaderOn;             { Set the window header on }
      SelectWorld(I);          { Select a world coordinate system }
      SetBackground(0);        { Give the window a black background }
      DrawBorder;              { Draw a border around  the window }
      DrawLines;               { Draw lines }
    end;

    repeat until KeyPressed;   { Wait until a key is pressed }

    LeaveGraphic;              { Leave the graphics system }

  end. { MultipleWindows }
```

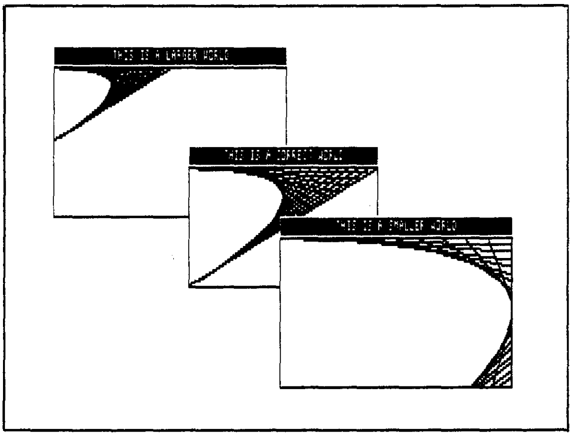


Figure 2-10 Three Windows (MULTWIND.PAS Example)

Moving Windows

Once you've defined a window, you can move it to any position on the screen using the *MoveVer* and *MoveHor* procedures; windows are moved by increments (multiples of 8 horizontal pixels and multiples of 1 vertical pixel).

MoveHor and *MoveVer* work by automatically and continually refreshing the screen images over which the window is moved. They do this by storing the displayed screen image to the virtual screen.

If you want to move multiple windows, things get a bit more complicated; you must manage the windows and other screen images yourself. What this means is that you must continually rebuild the virtual screen image every time you move windows. If there are any images on the screen that you wish to keep, you must copy those images either to the window stack with the *StoreWindow* procedure (if the images are in a window) or to the RAM (virtual) screen with the *CopyWindow* or *CopyScreen* procedure (if the images are on the screen) so they can be retrieved later; otherwise, when you move a window over those images, they will be erased, and there will be no way to restore them.

For your windows to keep their integrity and to be moved independently, you must keep copies of all windows on the window stack, and store all screen images you want to keep on disk. For instance, if the screen contains two windows that you want to display independently—that is, you want to be able to move them around and place them on top of each other—you should do the following: using the *SaveScreen* procedure, store the screen (without any windows) on disk, and store up-to-date copies of both windows on the window stack using the *StoreWindow* procedure.

Every time you draw something in a window, or change what was previously drawn, save a copy of the window on the window stack. When you want to move a window, save the presently displayed screen—without the window you plan to move—to the RAM virtual screen using the *CopyScreen* procedure, so the non-moving window is now also copied to the virtual screen. The virtual screen should now contain everything that was on the displayed screen, except the window you want to move. Now, draw the window you want to move on the screen, and use *MoveHor* and *MoveVer* to move the window around, without destroying the fixed images underneath.

The *window stack* is a RAM memory area where window images can be stored temporarily. You might want to use the stack when, for instance, you have defined and drawn in several windows but only want to display a few on the screen, or if one window is obstructing another and the obstructed window needs to be displayed. Whole window stacks, as well as individual windows in the stack, can be stored to and recalled from disk using the *SaveWindow* and *RestoreWindow* procedures. Windows on the stack can be accessed in any order.

Windows can be restored from the stack to any location on the screen by specifying *X* and *Y* offsets. To restore the window to its former position, use offsets of 0.

If the window currently selected with the *SelectWindow* procedure is the same as the one being restored from the stack, the screen coordinates of the selected window will shift to match the offset of the restored window. The selected window does *not* change when any other window is restored from the stack.

Stored windows and the RAM screen are dynamically allocated on the heap using the Turbo *GetMem* and *FreeMem* procedures. Therefore, the *Mark/Release* method of memory management should not be used in your programs.

The following program (MOVEWIND.PAS) shows how to move windows about on the screen; use the arrow keys to move the windows, and press the space bar to stop program execution.

```
program MoveWindows;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow;

const
  Null= #0;                  { The null character }

var
  I : integer;
  Ch : char;

procedure DrawLines;
var
  I : integer;
begin
  for I := 1 to 20 do
    DrawLine(I * 50, 0, 1000 - I * 50, I * 50);
end; { DrawLines }

begin { MoveWindows }

  InitGraphic;                { Init the graphics system }

  DrawBorder;                 { Draw a border around the drawing }
                              { area of the primary window }
                              { (the dimensions of the primary window }
                              { default to the screen dimensions) }

  DefineWindow(1, Trunc(XMaxGlb / 10), Trunc(YMaxGlb / 10),
                Trunc(XMaxGlb / 2), Trunc(YMaxGlb / 2));
                              { Define a window  one tenth of the way }
                              { in from the left and top edges, and half }
                              { way down from the right and bottom edges }

  DefineHeader(1, 'THIS IS THE FIXED WINDOW'); { Give it a header }
```

```
  DefineWorld(1, 0, 0, 1000, 1000);   { Give it a world coordinate system }

  DefineWindow(2, Trunc(XMaxGlb / 2), Trunc(YMaxGlb / 2),
                  Trunc((XMaxGlb * 9) / 10), Trunc((YMaxGlb * 9) / 10));
                            { Define a window  one half of the way }
                            { in from the left and top edges, and half }
                            { way down from the right and bottom edges }

  DefineHeader(2, 'THIS IS THE MOVEABLE WINDOW'); { Give it a header }

  DefineWorld(2, 0, 0, 1000, 1000);   { Give it a world coordinate system }

  SelectWindow(1);          { Select fixed window }
  SetHeaderOn;
  SelectWorld(1);           { Select its world }
  SetBackground(0);         { Give it a black background }
  DrawBorder;               { Draw a border around the window }
  DrawLines;                { Draw lines in it }
  CopyScreen;               { Copy it to the virtual screen }

  SetBreakOff;              { Don't error when edge hit }
  SetMessageOff;

  SelectWindow(2);          { Select moveable window }
  SetHeaderOn;
  SelectWorld(2);           { Select its world }
  SetBackground(0);         { Give it a black background }
  DrawBorder;               { Draw a border around the window }
  DrawLines;                { Draw lines in it }

  repeat
    Ch: = Readkey;          { Read the keystroke }

    if (Ch = Null) and KeyPressed then { Test for an extended scan code  }
      Ch: = Readkey;                   { on either an IBM or Zenith Z100 }
    case Ch of
      'A', 'H' : MoveVer(-4, true);   { Up arrow }
      'D', 'K' : MoveHor(-1, true);   { Left arrow }
      'C', 'M' : MoveHor(1, true);    { Right arrow }
      'B', 'P' : MoveVer(4, true);    { Down arrow }
    end;
  until Ch = ' ';           { Space character exits program }

  LeaveGraphic;             { Leave the graphics system }
end. { MoveWindows }
```

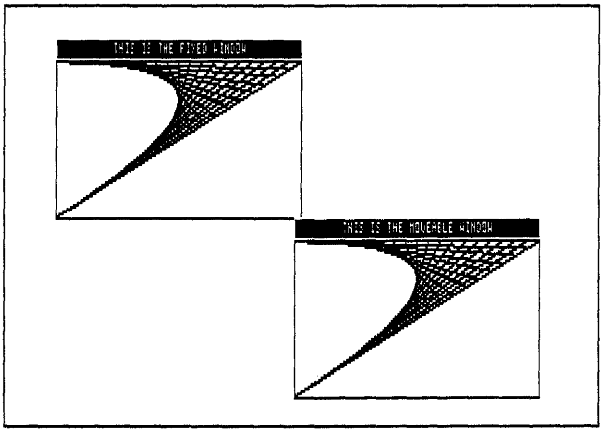


Figure 2-11 Moving a Window (MOVEWIND.PAS Example)

Another Use for Windows: The Flow Chart

Anything that can be contained in a rectangle can be animated using windows. The following example (FLOWDEMO.PAS) animates a flow chart by using a moving window. The drawing of the flow chart is the fixed screen image, while a window that contains the present state of the "machine" is moved along the flow chart drawing to show how the processor modifies variables when the program executes. The program increments a count and tests the result. If the count is not large enough, the program increments the count and tests again. When the count is high enough, the program is finished.

```
program FlowDemo;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow;

procedure FlowChartDemo;
var
  X1, Y1, X2, Y2, I, Count : integer;
  Temp : WrkString;

procedure DrawArrowHor(X1, Y1, X2, Y2 : integer);
{ Draw horizontal arrow with tip at point (X2, Y2) }
begin
  DrawLine(X1, Y1, X2, Y2);
  if X2 > X1 then
    begin
      DrawLine(X2 - 4, Y2 - 2, X2, Y2);
      DrawLine(X2 - 4, Y2 + 2, X2, Y2);
    end
  else
```

```
    begin
      DrawLine(X2 + 5, Y2 - 2, X2, Y2);
      DrawLine(X2 + 5, Y2 + 2, X2, Y2);
    end;
end; { DrawArrowHor }

procedure DrawArrowVer(X1, Y1, X2, Y2 : integer);
{ Draw vertical arrow with tip at point (X2, Y2) }
begin
  DrawLine(X1, Y1, X2, Y2);
  if Y2 > Y1 then
    begin
      DrawLine(X2 - 2, Y2 - 3, X2, Y2);
      DrawLine(X2 + 2, Y2 - 3, X2, Y2);
    end
  else
    begin
      DrawLine(X2 - 2, Y2 + 3, X2, Y2);
      DrawLine(X2 + 2, Y2 + 3, X2, Y2);
    end;
end; { DrawArrowVer }

procedure Blink(Count, Time : integer);
{ Blink the current window }
var
  I : integer;
begin
  for I := 1 to Count do
  begin
    Delay(Time);
    InvertWindow;
  end;
end; { Blink }

begin { FlowChartDemo }
  DefineWindow(1, 0, 0, 79, 185);         { Define the 'FLOW CHART' window }
  DefineWindow(2, 12, 20, 25, 40);        { Define the 'START' window }
  DefineWindow(3, 15, 55, 22, 75);        { Define the 'I=1' window }
  DefineWindow(4, 11, 110, 26, 130);      { Define the 'IF I<=5' window }
  DefineWindow(5, 47, 90, 56, 110);       { Define the 'I=I+1' window }

  ClearScreen;                            { Draw the surrounding window }
  SetColorWhite;
  DefineHeader(1, 'A FLOW CHART');
  SetHeaderOn;
  SelectWindow(1);
  DrawBorder;
  SetHeaderOff;

  SelectWindow(2);                        { Draw the 'START' window }
  DrawBorder;
  DrawText(125, 27, 2, 'START');
  SetWindowModeOff;
  DrawArrowVer(151, 40, 151, 55);         { Draw the connecting line }
  SetWindowModeOn;
```

```
SelectWindow(3);                    { Draw the 'I=1' window }
DrawBorder;
DrawText(136, 63, 2, 'I=1');
SetWindowModeOff;
DrawArrowVer(151, 75, 151, 110);    { Draw the connecting line }
SetWindowModeOn;

SelectWindow(4);                    { Draw the 'IF I>=5' window }
DrawBorder;
DrawText(108, 118, 2, 'IF I<=5');
DrawStraight(215, 417, 120);        { Draw the connecting lines }
SetWindowModeOff;
DrawArrowVer(417, 120, 417, 110);
DrawArrowVer(151, 130, 151, 155);
SetWindowModeOn;
SelectWindow(1);
DrawText(300, 110, 2, 'YES');
DrawText(160, 137, 2, 'NO');

SelectWindow(5);                    { Draw the 'I=I+1' window }
DrawBorder;
DrawText(390, 98, 2, 'I=I+1');
SetWindowModeOff;
DrawLine(417, 90, 417, 80);         { Draw the connecting lines }
DrawArrowHor(417, 80, 151, 80);

SetAspect(1.0);                     { Draw the 'END' circle }
DrawCircle(151, 165, 25);
SelectWindow(1);
DrawText(137, 163, 2, 'END');
SetWindowModeOn;
SetHeaderOn;

CopyScreen;                         { Make an image of this screen }
                                    { on the virtual RAM screen }

DefineWindow(2, 15, 21, 22, 39);    { Set up the moving window }
SelectWindow(2);
SetBackground(0);
DrawBorder;
InvertWindow;
Delay(1000);
InvertWindow;

Temp := '123456';                   { Init the number array, }
MoveVer(35, true);                  { move window over init statement, }
DrawText(139, 63, 2, 'I=' + Temp[1]); { 'init' it }
Blink(30, 50);
MoveVer(55, true);                  { Move it down to increment loop }

for Count := 2 to 6 do              { Do increment loop }
```

```
    begin
      Delay(500);
      MoveHor(33, true);
      MoveVer(-20, true);
      SetBackground(0);
      DrawBorder;
      DrawText(400, 98, 2, 'I=' + Temp[Count]);
      Blink(30, 50);
      MoveVer(-20, true);
      MoveHor(-33, true);
      MoveVer(40, true);
    end;

    InvertWindow;
    Delay(1000);
    MoveVer(46, true);                     { Move to the 'END' statement }
    Blink(30, 50);

    MoveHor(45, true);                     { Move back up to the top }
    MoveVer(-136, true);
    MoveHor(-45, true);
    SetHeaderOn;
  end; { FlowChartDemo }

  begin
    InitGraphic;                           { Init the graphics system }

    FlowChartDemo;                         { Do the demo }

    repeat until KeyPressed;               { Wait until a key is pressed }

    LeaveGraphic;                          { Leave the graphics system }
  end. { FlowDemo }
```

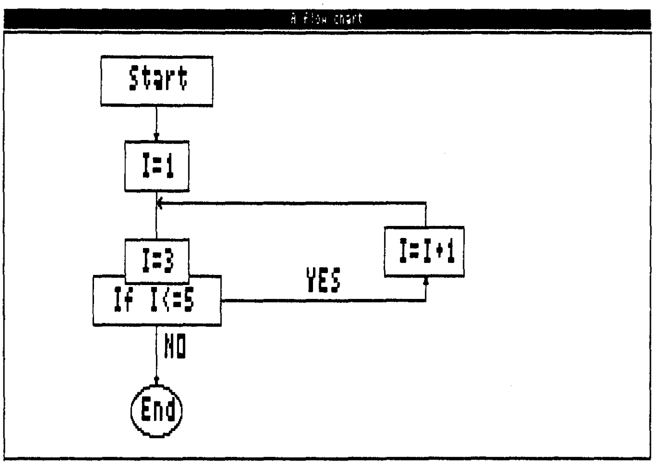


Figure 2-12 A Flow Chart (FLOWDEMO.PAS Example)

Summary of Window Routines

- *ClearWindowStack* deletes a window from the stack.
- *CopyScreen* copies the active screen onto the inactive screen.
- *CopyWindow* copies a window from one screen to another.
- *DefineHeader* defines a window header.
- *DefineWindow* defines an area of the screen as a window.
- *DefineWorld* defines a world coordinate system.
- *DrawBorder* draws a line around the window.
- *GetWindow* returns the code number of the active window.
- *InvertWindow* inverts the color of the active window.
- *LoadWindow* loads a window from disk to the specified world coordinates.
- *LoadWindowStack* stores a window stack from disk to the window stack.
- *RedefineWindow* changes the dimensions of an existing window.
- *RemoveHeader* removes a window header.
- *ResetWindowStack* erases all windows from the stack.
- *ResetWindows* sets all windows to the size of the physical screen.
- *SaveWindow* saves a window to disk.
- *SaveWindowStack* saves a window stack to disk.
- *SelectWindow* selects a window for drawing.
- *SelectWorld* selects a world coordinate system.
- *SetHeaderOff* and *SetHeaderOn* determine whether a window header is displayed.
- *SetHeaderToBottom* and *SetHeaderToTop* place a header at the bottom or top of a window.
- *SetWindowModeOff* and *SetWindowModeOn* determine whether drawing takes place in a window or on the screen.
- *Store Window* stores a window on the window stack.
- *WindowMode* returns the window status.
- *WindowSize* determines whether there is room for a window on the stack.
- *WindowStackSize* returns the number of free bytes on the window stack.

Pie and Bar Charts

Pie and bar charts provide a way to graphically represent numeric results that are common to many business and statistical applications. Three high-level routines—*DrawCartPie* and *DrawPolarPie* for pie charts, and *DrawHistogram* for bar charts—do most of the work required to display information in pie and bar charts; all you have to do is supply the numerical data. As long as you are familiar with Turbo Pascal, the program examples used in this section can be easily tailored to a particular application.

Pie Charts

Pie charts are used to display a series of values or percentages (the pie "slices") that make up a total unit (the whole pie). A pie chart shows, at a glance, the relative proportion of the whole that is represented by each value. For instance, a pie chart format is an effective way to show a company's market share, or the results of a scientific experiment.

The *DrawCartPie* and *DrawPolarPie* procedures not only automatically draw a pie chart that corresponds to your input values; they can also label each pie segment with text and/or a numeric value, as well as pull any pie segment away from the pie for display emphasis. Although pie charts can be drawn with reference to either world or screen coordinates, it is usually best to use world coordinates, especially if you want your program to run correctly on different computer systems. Also, pie charts drawn using a world coordinate system will be correctly proportioned in any given window, regardless of the size of the window.

A pie chart is drawn by passing the following parameters:

- Coordinates of the center point of the pie
- Coordinates of the starting point of the first pie segment
- Value and optional label of each segment in an array
- Desired labeling options
- Scale of the label characters (multiples of 4x6 pixels)

A pie chart can be specified so that the starting point of the first segment of the pie chart is referenced to either of two coordinate systems: Cartesian coordinates [*X*,*Y*], or polar coordinates [*Radius*,*Angle*]. The *Cartesian coordinate system*, used by the *DrawCartPie* procedure, allows the drawing to be referenced to a position located by [*X*,*Y*] coordinates. For instance, the first pie segment can be defined by a point relative to the center of the pie. The *polar coordinate system* references the pie chart to its radius and the angle of its first segment.

It is usually easiest to use polar coordinates—that is, to think of a pie chart as a circle with a certain radius, and with its first segment starting at a particular angle. The *DrawPolarPie* procedure uses polar coordinates. Since this is the method used most often, the *DrawPolarPie* procedure is used in the example program ONEPIE.PAS.

In this example, *DrawPolarPie* first defines a window that is the size of the entire screen, with a header and border. Next, the array of values and optional text labels to be used in the creation of the pie chart are initialized. This part of the example is normally the only part that is application-specific. The size of each pie segment is specified by the *area* entry in this array. This area is displayed as a percentage of the total area (determined by totalling all the other areas to be displayed in the pie). The numbers appropriate to your application are used here, and the *DrawPolarPie* procedure displays each segment according to its percentage of the whole pie. If you give any of the array entries a negative value, the pie drawing procedure will move this segment outward. This feature can be used to draw attention to important segments of the pie chart.

The *Mode* parameter allows you to display area values and/or text contained in *PieArray* as labels. These labels are usually displayed at the end of optional label lines. The area information is displayed exactly as passed in the array. If you don't want to display the numeric value of the segment, the *Mode* parameter allows you to display a text label only; the text is passed in the *PieArray*. The text label can include any alphanumeric character or ESCape sequence (used to specify special graphics characters). See the *DrawCartPie* and *DrawPolarPie* procedures in Chapter 3 for more information about this option.

The next part of the ONEPIE.PAS example determines the position, size, and shape of the pie to be drawn. The pie is specified by the coordinates of its center point, and radius and starting angle. (If the example were using the *DrawCartPie* procedure, the starting point would be specified by an [X,Y] position.)

The shape of the pie chart, like any other circle, is determined by its aspect ratio—its height-to-width ratio. You can vary the shape of the pie chart by calling the *SetAspect* procedure. In addition, a global constant, *AspectFactor*, sets the system-dependent aspect ratio, so that an aspect ratio of 1.0 produces a true circle on a particular hardware screen.

The parameters *InRadius* and *OutRadius* specify the inside and outside endpoints of the radial label line. This label line relates a text and numeric label with a particular pie segment. *InRadius* and *OutRadius* are referenced to the edge of the pie chart. A value of 1.0 puts the endpoint on the edge of the pie chart, a value of 0.5 puts the endpoint halfway between the edge and the center, and a value of 2.0 puts the endpoint at a distance of twice the radius out from the center of the chart. If both *InRadius* and *OutRadius* are 1.0, the label line is one dot long, coincides with the edge of the pie chart, and, thus, for all practical purposes, is not drawn.

The final parameters, *Mode* and *Size*, specify which labels, if any, are drawn, and their size. *Mode* allows four possibilities: no label, text label only, numeric label only, and both text and numeric label. *Size* specifies the scale of the label characters (multiples of 4x6 pixels).

```
program OnePieDemo;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow, GShell;

procedure OnePieDem;

var
  Sum, X1, Y1, Radius, Theta, InRadius, OutRadius : Float;
  N, Mode, Size : integer;
  A : PieArray;
  Back : byte;
  Ch : char;

begin { OnePieDem }
  ClearScreen;
  SetColorWhite;
  DefineWindow(1, 0, 0, XMaxGlb, YMaxGlb);
  DefineHeader(1, 'A SINGLE PIE CHART'); { Set up a window }
  DefineWorld(1, 0, 0, 1000, 1000);
  SelectWorld(1);
  SelectWindow(1);
  SetHeaderOn;
  SetBackground(0);
  DrawBorder;

  N := 5;                               { The number of pie segments }
  A[1].Area := 25;                      { Init the pie array }
  A[2].Area := 17.5;
  A[3].Area := 9.6;
  A[4].Area := 21;
  A[5].Area := 35;
  A[1].Text := 'JAN.=';
  A[2].Text := 'FEB.=';
  A[3].Text := 'MAR.=';
  A[4].Text := 'APR.=';
  A[5].Text := 'MAY=';

  A[1].Area := -A[1].Area;              { Move the first segment outward }

  X1 := 500;                            { Set the center to mid-screen }
  Y1 := 500;

  Radius := 200;                        { Set the start of the circle }
  Theta := 60;

  SetAspect(1.0);                       { Set the aspect ratio }

  InRadius := 0.7;                      { Set the ends of the label line }
  OutRadius := 1.25;

  Mode := 2;                            { Set to draw both lables }
  Size := 2;                            { Set to text size 2 }
```

```
                                          { Draw the pie }
  DrawPolarPie(X1, Y1, Radius, Theta, InRadius,
               OutRadius, A, N, Mode, Size);
end; { OnePieDem }

begin { OnePieDemo }
  InitGraphic;                            { Init the graphics system }

  OnePieDem;                              { Do the demo }

  repeat until KeyPressed;                { Wait until a key is pressed }

  LeaveGraphic;                           { Leave the graphics system }
end. { OnePieDemo }
```

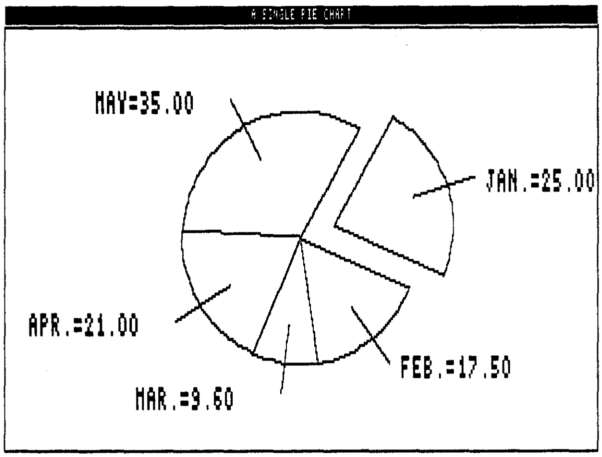


Figure 2-13 A Pie Chart (ONEPIE.PAS Example)

Bar Charts

Bar charts (histograms) are used to represent the way a given set of data changes over time. A bar chart displays a sequence of values as vertical bars, with each bar's height in proportion to the value associated with that bar. A bar chart is automatically generated by passing the array of the values to be displayed to the *DrawHistogram* procedure. The resulting bars are drawn adjacent to each other and always completely fill the width of the active window. The height of the bars is scaled by the world coordinate system active at the time.

The demo program ONEHIST.PAS is an example of the *DrawHistogram* procedure. This program draws ten random-length bars across a window that fills the screen. The procedure first fills the *PlotArray* with ten random values. The *PlotArray* is the same as that used to plot polygons, and therefore has two values in

each array element (the *X* position and the *Y* displacement). The *X* value in this case is determined by the program, and the *Y* (vertical displacement) value is used by *DrawHistogram*.

Next, a window is defined and selected that fills the screen, and a world coordinate system is defined and selected that will determine the vertical scaling of the bar lengths. The *X* dimension specification can take any value except 0, since it is corrected for by the *DrawHistogram* routine.

The *Hatch* parameter specifies whether the bars are to be cross-hatched with slanting lines; *HatchDen* specifies the number of vertical pixels between each hatch line. The sign of *HatchDen* determines the direction of hatching; if it is negative, the first hatch line goes from lower left to upper right (positive slope); if it is positive, the first hatch line goes from upper right to lower left (negative slope); hatching direction alternates with each consecutive bar. In the call to *DrawHistogram*, a negative value for *DisplyLen* indicates that bars should be drawn from the *Y* zero axis (which is, in this case, at the bottom of the window). A positive value would specify that bars are always drawn from the bottom of the window, with negative values plotted as positive values.

```
program OneHist;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow, GShell;

procedure HistoDem;
var
  I, DisplyLen, HatchDen : integer;
  A : PlotArray;
  R : Float;
  Ch : char;
  Hatch : boolean;

begin
  DisplyLen := 10;                  { Draw 10 bars }

  for I := 0 to DisplyLen do        { Init the display array with random #'s }
  begin;
    A[I+1, 2] := Random;
  end;

  SetColorWhite;                    { Set up the window for the bar chart }
  SetBackground(0);
  SetHeaderOn;
  DefineWindow(1, 0, 0, XMaxGlb, YMaxGlb);
  DefineHeader(1, 'A RANDOM BAR CHART WITH HATCHING');
  DefineWorld(1, -10, 0, 10, 1.0);
  SelectWorld(1);
  SelectWindow(1);

  DrawBorder;                       { Draw the window }

  Hatch := true;                    { Enable hatching }
  HatchDen := 7;                    { Draw hatch lines this far apart }
```

```
  DrawHistogram(A, -DisplyLen, Hatch, HatchDen);   { Draw the bar chart }
end; { HistoDem }

begin { OneHist }
  InitGraphic;                        { Init the graphics system }

  HistoDem;                           { Do the demo }

  repeat until KeyPressed;            { Wait until a key is pressed }

  LeaveGraphic;                       { Leave the graphics system }
end. { OneHist }
```

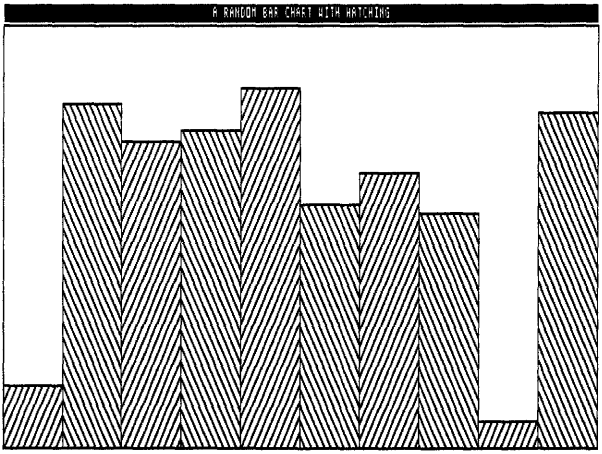


Figure 2-14   A Bar Chart (ONEHIST.PAS Example)

The following example program (PIEHISTO.PAS on the distribution disk) shows both a bar and pie chart displaying the same data. An example of how to label bar charts is also included.

```
program PieHisto;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow, GShell;

procedure PieHistoDem;
var
  Sum, X1, Y1, X2, Y2, InRadius, OutRadius : Float;
  I, N : integer;
  A : PieArray;
  B : PlotArray;
  Ch : char;
  NumText : WrkString;
```

```
begin
  N := 5;                               { The number of data points }
  A[1].Area := 25;                      { Init the pie array }
  A[2].Area := 17.5;
  A[3].Area := 9.6;
  A[4].Area := 21;
  A[5].Area := 35;
  A[1].Text := 'JAN. ';
  A[2].Text := 'FEB. ';
  A[3].Text := 'MAR. ';
  A[4].Text := 'APR. ';
  A[5].Text := 'MAY  ';

  for I := 1 to N do                     { Init the histogram array }
    B[I, 2] := A[I].Area;

  ClearScreen;
  SetColorWhite;

  DefineWindow(1, 0, 0, XMaxGlb, YMaxGlb);
  DefineHeader(1, 'BOTH A PIE AND A BAR CHART');  { Set up a window }
  SelectWindow(1);
  SetHeaderOn;
  SetBackground(0);
  DrawBorder;

  for I := 1 to N do                     { Type the info in the upper-right
                                           corner }
  begin
    GotoXY(60, 4 + I);                   { Goto correct line }
    Write(A[I].Text, '=');               { Type the label info }
    Str(A[I].Area:6:2, NumText);         { Format the numeric info }
    Write(NumText);                      { Type the numeric info }
  end;

  DefineWindow(2, Trunc(XMaxGlb / 10), Trunc(YMaxGlb / 10),
                  Trunc(XMaxGlb * 6 / 10), Trunc(YMaxGlb * 7 / 10));
  DefineHeader(2, 'A PIE CHART');        { Set up a window }
  DefineWorld(2, 0, 0, 1000, 1000);
  SelectWorld(2);
  SelectWindow(2);
  SetHeaderOn;
  SetBackground(0);
  DrawBorder;

  A[1].Area := -A[1].Area;              { Move the first segment outward }
  SetAspect(1.0);                       { Set the aspect ratio }

  X1 := 500;                            { Set the center to mid-screen }
  Y1 := 500;

  X2 := 600;                            { Set the start of the circle }
  Y2 := 350;

  InRadius := 0.7;                      { Set the ends of the label line }
  OutRadius := 1.25;

  DrawCartPie(X1, Y1, X2, Y2, InRadius, OutRadius, A, N, 2, 1); { Draw the pie }
```

```
  A[1].Area := -A[1].Area;                { Reset the sign }

  DefineWindow(3, Trunc(XMaxGlb / 2), Trunc(YMaxGlb / 2),
                  Trunc(XMaxGlb * 9 / 10), Trunc(YMaxGlb * 9 / 10));
  DefineHeader(3, 'A BAR CHART');         { Set up a window }
  DefineWorld(3, 0, 0, 10, 60);
  SelectWorld(3);
  SelectWindow(3);
  SetHeaderOn;
  SetBackground(0);
  DrawBorder;

  DrawHistogram(B, N, true, 5);

  for I := 1 to N do                      { Draw the bar chart lables }
  begin
    DrawTextW((10 / N) * (I - 1), 10, 1, '   ' + A[I].Text); { Draw the text }
    Str(A[I].Area:6:2, NumText);                          { Format the number }
    DrawTextW((10 / N) * (I - 1), 16, 1, ' ' + NumText);    { Draw the number }
  end;
end;  { PieHistoDem }

begin { PieHisto }
  InitGraphic;                            { Init the graphics system }

  PieHistoDem;                            { Do the demo }

  repeat until KeyPressed;                { Wait until a key is pressed }

  LeaveGraphic;                           { Leave the graphics system }
end. { PieHisto }
```

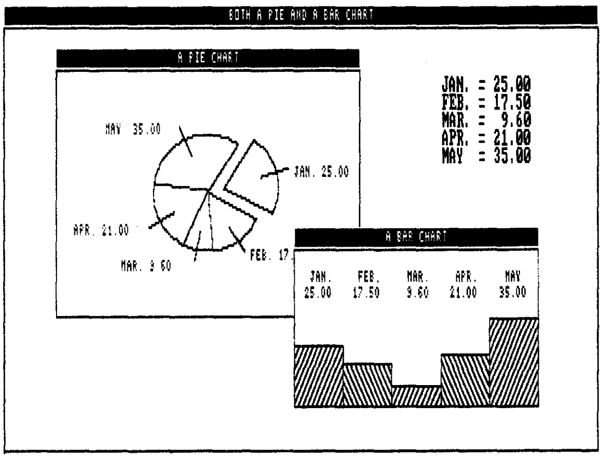


Figure 2-15 Pie and Bar Chart Displaying Same Data (PIEHISTO.PAS Example)

Summary of Pie and Bar Chart Routines

- *DrawCircleSegment* draws an arc of a circle with optional text and numeric labels.
- *DrawCartPie* draws a pie chart using Cartesian coordinates.
- *DrawPolarPie* draws a pie chart using polar coordinates.
- *DrawHistogram* draws a bar chart.

Plotting Curves

Any curve that is made up of a series of points, usually connected by line segments, is called a *polygon*. By default (constant *MaxPlotGlb*), a polygon consists of a maximum of 100 points. If your application requires more points, the default for *MaxPlotGlb* can be changed, or, alternatively, multiple polygons can be used to create the final image. Because the resolution of a screen is limited, a smooth curve can usually be made out of a small number of line segments.

A Simple Example: Plotting a Sine Curve

The following example program (ONEPOLY.PAS on the distribution disk) plots a smooth sine curve using the *DrawPolygon* procedure. This example uses 30 line segments to produce the curve. In this case, the full screen is used; on the standard IBM graphics screen, there are approximately 20 pixels between the endpoints of the line segments. As can be seen from the graphics display output by this program, a smooth curve is drawn when this number of segments is used, with little evidence of where one line segment ends and another begins.

DrawPolygon receives its input as *X* and *Y* coordinates in the array *PlotArray*. This array is two dimensional—that is, each point's sequence in the curve is specified by the first dimension, and its *X* and *Y* values are selected by the second dimension. For example the value in the array *PlotArray*[5,1] would be the *X* position of the 5th point, and the value in *PlotArray*[5,2] would be the *Y* position. A symbol can be optionally placed at each vertex point on the curve. If the value used to specify the symbol type is negative, the symbols are not connected by lines. The size of the symbols, and whether lines are drawn from the vertices to the *X* axis, are also specified by parameters passed to the procedure. See the *DrawPolygon* procedure description in Chapter 3 for detailed information about these options. This example simply draws a single sine curve across the screen. The array to draw is passed to *DrawPolygon* (with instructions to draw from the first to last point in the

array) with no symbols at the vertices, and the curve is drawn as a series of line segments that connect the vertices.

The *PlotArray* for *DrawPolygon* is filled by the *GenerateFunction* procedure. Varying the value of *n* in this program varies the number of vertices in the curve. You can use the ONEPOLY.PAS example to experiment with the proper number of points needed to generate a smooth curve on your screen. In addition, you can draw a subset of the polygon by starting and ending the drawing on any element of this array; the indices of the desired start and end points are passed to the routine as parameters.

```
program OnePolygon;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow, GShell;

procedure PolygonDem;
var
  N : integer;
  B, A : PlotArray;
  Ch : char;
  X1, X2 : integer;

procedure GenerateFunction(var A : PlotArray; N : integer);
{ Generate a sine polygon }
var
  I : integer;
  Delta : Float;
begin
  Delta := 2 * Pi / (N - 1);
  for I := 1 to N do
  begin
    A[I, 1] := (I - 1) * Delta - Pi;
    A[I, 2] := Sin(A[I, 1]);
  end;
end;  { GenerateFunction }

begin
  ClearScreen;

  N := 30;
  GenerateFunction(A, B, N);            { Generate the polygon }

  DefineWindow(1, 0, 0, XMaxGlb, YMaxGlb);
  DefineHeader(1, 'SINE CURVE AS A POLYGON');  { Set up the screen }
  DefineWorld(1, -Pi, -1, Pi, 1);
  SelectWorld(1);
  SelectWindow(1);
  SetBackground(0);
  SetHeaderOn;
  DrawBorder;

  DrawPolygon(A, 1, N, 0, 0, 0);        { Draw the polygon }
end;  { PolygonDem }
```

```
begin
  InitGraphic;                          { Init the graphics system }

  PolygonDem;                           { Do the demo }

  repeat until KeyPressed;              { Wait until a key is pressed }

  LeaveGraphic;                         { Leave the graphics system }
end. { OnePolygon }
```

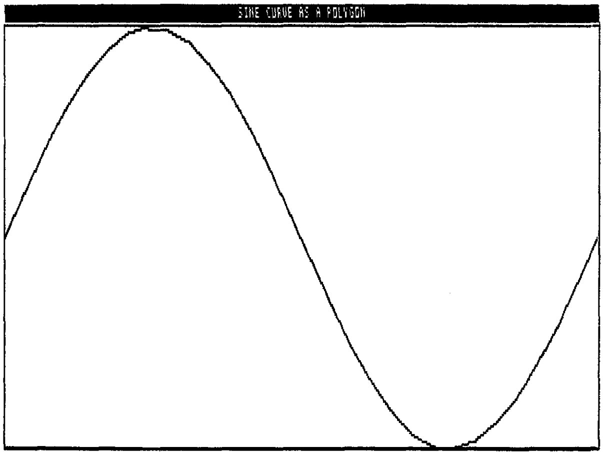


Figure 2-16 *Plotting a Smooth Curve (ONEPOLY.PAS Example)*

The DrawAxis Procedure

In many graphics applications that illustrate numeric results, it is useful to display a ruler that indicates the values of the displayed results. The *DrawAxis* procedure is used to draw rulers (and accompanying *X* and *Y* axes) along the left and bottom edges of the area that contains the graph. The rulers are scaled to fit the active window. *DrawAxis* automatically creates a new subwindow, bounded by the rulers, where drawing will take place. The world coordinate system (defined by the *DefineWorld* procedure) now fits in this subwindow.

Optional parameters passed to *DrawAxis* can provide a space between the rulers and the active window boundaries. This feature can also be used to provide space between legends or axis labels and the rulers, and/or to display multiple axes in one window. Other options can draw a border around the subwindow, turn the display of numeric labels and ruler tick marks on and off, draw zero *X* and *Y* axes, and select a line style for the axes.

The rulers have a couple of characteristics you should understand if you are to use them effectively. First, and most important, ruler markings are spaced according to screen pixel spacing. This means that the numbers associated with the rulers are correct with respect to the curve, but do not necessarily mark the decimal (or other number system) locations relevant to your application. In other words, ruler labels do not necessarily increment by one, ten, or other standard unit. In addition, with a higher resolution screen, (such as with the Hercules card), there will be more markings than with the same rulers drawn using a standard IBM graphics card.

The following example (ONEAXIS.PAS on the distribution disk) shows the simplest use of the *DrawAxis* procedure. This example defines a window that fills the whole screen, defines a world, and draws coordinate axes for the whole screen.

```
program OneAxis;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow, GShell;

procedure OneAxisDem;
begin
  ClearScreen;                        { Init screen }
  SetColorWhite;
  SetBackground(0);

  DefineHeader(1, 'LABELED AXES');    { Define the window }
  SetHeaderOn;
  DefineWorld(1, -10, -10, 10, 10);
  SelectWindow(1);
  SelectWorld(1);

  DrawBorder;                         { Draw it }
  DrawAxis(8, -7, 0, 0, 0, 0, 0, 0, true); { Draw coordinate axes }
end;

begin
  InitGraphic;                        { Initialize the graphics system }
  OneAxisDem;                         { Do the demo }
  repeat until KeyPressed;            { Wait until a key is pressed }
  LeaveGraphic;                       { Leave the graphics system }

end.
```

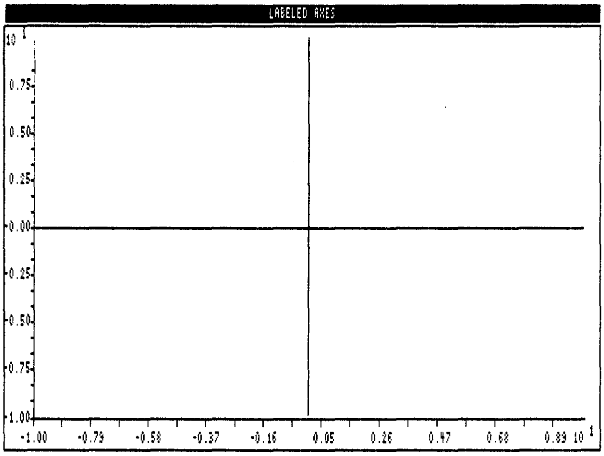


***Figure 2-17** Labeled Axes (ONEAXIS.PAS Example)*

Drawing a Sine Curve with Axes

The following example (POLYAXIS.PAS on the distribution disk) combines the previous two examples to display sine curve inside axes that are bounded by the screen edges.

```
program OnePolygon;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow, GShell;

procedure PolygonDem;
var N : integer;
  B, A : PlotArray;
  Ch : char;
  X1, X2 : integer;

procedure GenerateFunction(var A, B : PlotArray; N : integer);
{ Generate a sine polygon }
var I : integer;
  Delta : Float;
begin
  Delta := 2 * Pi / (N - 1);
  for I := 1 to N do
  begin
    A[I, 1] := (I - 1) * Delta - Pi;
    A[I, 2] := Sin(A[I, 1]);
  end;
end;  { GenerateFunction }
```

```
begin
  ClearScreen;

  N := 30;
  GenerateFunction(A, B, N);              { Generate the polygon }

  DefineWindow(1, 0, 0, XMaxGlb, YMaxGlb);
  DefineHeader(1, 'SINE CURVE AS A POLYGON');    { Set up the screen }
  DefineWorld(1, -Pi, -1, Pi, 1);
  SelectWorld(1);
  SelectWindow(1);
  SetBackground(0);
  SetHeaderOn;
  DrawBorder;

  DrawAxis(8, -8, 0, 0, 0, 0, 0, 0, false);{ Draw the axes }

  DrawPolygon(A, 1, N, 0, 0, 0);          { Draw the polygon }
end;  { PolygonDem }

begin
  InitGraphic;                            { Init the graphics system }

  PolygonDem;                             { Do the demo }

  repeat until KeyPressed;                { Wait until a key is pressed }

  LeaveGraphic;                           { Leave the graphics system }
end. { OnePolygon }
```

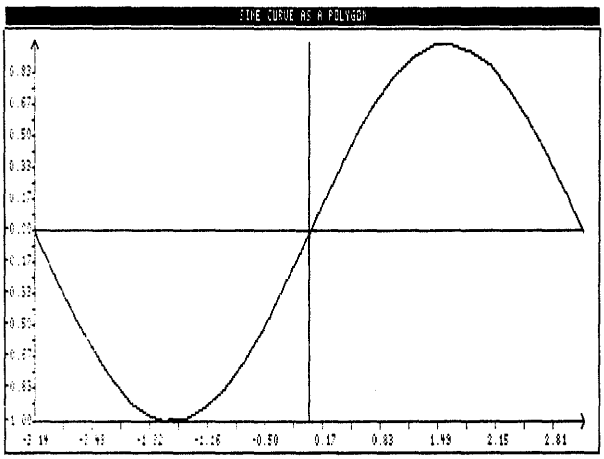


Figure 2-18   A Smooth Curve and Coordinate Axes (POLYAXIS.PAS Example)

There are several procedures that adjust the values in the *PlotArray* to *translate* (move), or rotate a polygon. These routines could be used for animation applications, to allow a single polygon to be used as the model for all the polygons that are to be subsequently moved about on the screen.

The example program (MOVEPOLY.PAS on the distribution disk) uses the *RotatePolygon* and *TranslatePolygon* procedures to draw an arrowhead on the screen, enable the cursor keys to rotate it, and move it forward and backward in the direction pointed to by the arrow. To end program execution, press the space bar.

The program initializes the polygon as an arrowhead in the center of the world, pointing towards the top of the screen. *RotatePolygon* rotates the polygon around its present "center of mass." This means that the polygon rotates around itself, rather than the origin (point [0,0]) of the coordinate system. To rotate the polygon about the origin (or any other point), use the *RotatePolygonAbout* procedure.

The *TranslatePolygon* procedure is used to move the arrowhead in the direction it is pointing. When the polygon is rotated, new increment values are used to translate the polygon in the new direction.

To move a polygon, you must first erase the old image before redrawing the new one. To do this, set the drawing color to black with the *SetColorBlack* procedure before calling *DrawPolygon* with the information from the last polygon.

There are no limits on where the polygon can be moved. Since the polygon is positioned using real coordinates, it would take a long time for this program to move the object to the end of the real number system. However, it does not take long to move the arrowhead off the screen. To make the program display the polygon in the world correctly, and to prevent the arrow from moving off the screen and destroying part of program memory, this program activates clipping by assigning a negative value to the last point to be displayed when calling *DrawPolygon*. When the arrowhead goes off the screen, *DrawPolygon* only draws the part of the line that fits the defined world.

```
program MovePolygon;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow, GShell;

var
  ArrowAngle : integer;
  Ch : char;
  Arrow : PlotArray;
  CurrX, CurrY, IncrX, IncrY, Size, Speed : Float;
  ArrowIncr : array[0..7, 1..2] of Float;
```

```
procedure MakeArrow;
begin
  Arrow[1, 1] := 0;                      { PlotArray init for the arrowhead }
  Arrow[1, 2] := 0;
  Arrow[2, 1] := Size;
  Arrow[2, 2] := -Size;
  Arrow[3, 1] := 0;
  Arrow[3, 2] := Size;
  Arrow[4, 1] := -Size;
  Arrow[4, 2] := -Size;
  Arrow[5, 1] := 0;
  Arrow[5, 2] := 0;
end;  { MakeArrow }

procedure MakeMoveTable;
begin
  ArrowIncr[0, 1] := 0;                  { Component velocities for radial moves }
  ArrowIncr[0, 2] := 1;
  ArrowIncr[1, 1] := -1;
  ArrowIncr[1, 2] := 1;
  ArrowIncr[2, 1] := -1;
  ArrowIncr[2, 2] := 0;
  ArrowIncr[3, 1] := -1;
  ArrowIncr[3, 2] := -1;
  ArrowIncr[4, 1] := 0;
  ArrowIncr[4, 2] := -1;
  ArrowIncr[5, 1] := 1;
  ArrowIncr[5, 2] := -1;
  ArrowIncr[6, 1] := 1;
  ArrowIncr[6, 2] := 0;
  ArrowIncr[7, 1] := 1;
  ArrowIncr[7, 2] := 1;
end;  { MakeMoveTable }

procedure MoveForward;                   { Routine to move polygon forward }
begin
  SetColorBlack;                         { Draw over old polygon to erase it }
  DrawPolygon(Arrow, 1, -5, 0, 0, 0);
  CurrX := CurrX + IncrX;                { Move to new position }
  CurrY := CurrY + IncrY;
  TranslatePolygon(Arrow, 5, IncrX, IncrY);
  SetColorWhite;                         { Draw polygon in new position }
  DrawPolygon(Arrow, 1, -5, 0, 0, 0);
end;  { MoveForward }

procedure MoveBack;                      { Routine to move polygon back }
begin
  SetColorBlack;                         { Same as above }
  DrawPolygon(Arrow, 1, -5, 0, 0, 0);
  CurrX := CurrX - IncrX;
  CurrY := CurrY - IncrY;
  TranslatePolygon(Arrow, 5, -IncrX, -IncrY);
  SetColorWhite;
  DrawPolygon(Arrow, 1, -5, 0, 0, 0);
end;  { MoveBack }
```

```
procedure TurnLeft;                    { Rotate polygon counterclockwise }
begin
  SetColorBlack;                       { Erase old polygon }
  DrawPolygon(Arrow, 1, -5, 0, 0, 0);
  RotatePolygon(Arrow, 5, 45);         { Rotate it 45 degrees }
  ArrowAngle := ArrowAngle + 1;
  if ArrowAngle > 7 then
    ArrowAngle := 0;
  IncrX := Speed * ArrowIncr[ArrowAngle, 1];  { Get new velocity }
  Incry := Speed * ArrowIncr[ArrowAngle, 2];
  SetColorWhite;                       { Draw rotated polygon }
  DrawPolygon(Arrow, 1, -5, 0, 0, 0);
end;  { TurnLeft }

procedure TurnRight;                   { Rotate polygon clockwise }
begin
  SetColorBlack;                       { Same as above }
  DrawPolygon(Arrow, 1, -5, 0, 0, 0);
  RotatePolygon(Arrow, 5, -45);
  ArrowAngle := ArrowAngle - 1;
  if ArrowAngle < 0 then
    ArrowAngle := 7;
  IncrX := Speed * ArrowIncr[ArrowAngle, 1];
  Incry := Speed * ArrowIncr[ArrowAngle, 2];
  SetColorWhite;
  DrawPolygon(Arrow, 1, -5, 0, 0, 0);
end;  { TurnRight }

begin
  InitGraphic;                         { Init the graphics system }

  DefineWindow(1, 0, 0, XMaxGlb, YMaxGlb);

  DefineWorld(1, -1000, -1000, 1000, 1000); { Give it a world coordinate system }

  SelectWorld(1);                      { Select its world }
  SelectWindow(1);                     { Select window }
  SetBackground(0);                    { Give it a black background }

  Size := 100;
  Speed := 30;
  CurrX := 0;
  CurrY := 0;
  ArrowAngle := 0;
  IncrX := 0;
  IncrY := Speed;

  MakeArrow;                           { Make the arrowhead }
  MakeMoveTable;                       { Make the move table }
  DrawPolygon(Arrow, 1, 5, 0, 0, 0);   { Draw it pointing up }
```

```
  repeat
    Ch: = ReadKey;                        { Read the keystroke }

    if (Ch = #0) and KeyPressed then   { Test for an extended scan code  }
      Ch: = ReadKey;                    { on either an IBM or Zenith Z100 }
    case Ch of
      'A', 'H' : MoveForward;          { Up arrow }
      'D', 'K' : TurnLeft;             { Left arrow }
      'C', 'M' : TurnRight;            { Right arrow }
      'B', 'P' : MoveBack;             { Down arrow }
    end;
  until Ch = ' ';                       { Space character exits program }

  LeaveGraphic;                         { Leave the graphics system }
end. { MovePolygon }
```

Finding a World to Fit a Polygon

In many applications that involve curves, the final form of the graph that is to be displayed is not known until the program is run. In these cases, the *FindWorld* procedure can be used to find the world coordinate system that will exactly fit the curve, or that is a specified percentage larger than the curve. *FindWorld* ensures that the area in which your curve is displayed is of the proper dimensions for your application. This procedure, in conjunction with the *DrawAxis* and *DrawPolygon* procedures, can produce a tailormade graphic presentation.

The *FindWorld* procedure always sets up a world with its lowest coordinates at the upper left corner of the window. The following code can be used after a call to *FindWorld* to turn the world coordinate system upside down:

```
with World[I] do   {integer I is the world being changed}
 begin
  Temp:=Y1;        {Temp is a real variable}
  Y1:=Y2;
  Y2:=Temp;
end;
```

This must be done *before* selecting the world! (You can also flip the coordinate system horizontally by swapping the *X* coordinates *X1* and *X2*.)

Note: World coordinates in earlier versions of the Turbo Pascal Graphix Toolbox defined the *Y* axis using the Cartesian coordinate system. If you have a program written for an earlier version, you must switch the two *Y* parameters in each instance of *DefineWorld*, so that, for example,

```
DefineWorld(WorldNumber,X1,Y1,X2,Y2);
```

becomes

```
DefineWorld(WorldNumber,X1,Y2,X2,Y1);
```

The following program (FINDWRLD.PAS on the distribution disk) demonstrates *FindWorld* and some of the more advanced features of the *DrawPolygon* and *DrawAxis* routines. This program draws five random points on the screen with star symbols at each of the vertices, and axis rulers to show the scale of the numbers.

```
program FindWorld;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow, GShell;

procedure FindWorldDem;
var
  X : Float;
  Dx, Dy, I, N, Lines, Scale : integer;
  X1, Y1, X2, Y2 : integer;
  B, A : PlotArray;

begin
  DefineWindow(1, 0, 0, XMaxGlb, YMaxGlb); { Define windows as whole screen }
  DefineWindow(2, 0, 0, XMaxGlb, YMaxGlb);
  DefineWorld(1, 0, 0, 1000, 1000);        { Give a world to the screen }

  DefineHeader(2, 'A FOUND WORLD');        { Window where curve will go }
  SelectWindow(2);
  SetHeaderOn;

  N := 10;                                 { Fill polygon array }
  for I := 1 to N do
  begin
    A[I, 1] := I - 1;
    A[I, 2] := random - 0.5;
  end;

  FindWorld(2, A, N, 1, 1.08);             { Make world 2 the right size }

  SelectWindow(2);                         { Select it and draw border }
  DrawBorder;

  Dx := -8;                                { Draw axis inset from window edge }
  Dy := 7;
  X1 := 3;
  Y1 := 5;
  X2 := 25;
  Y2 := 10;
  Lines := 0;
  Scale := 0;

  SetLineStyle(0);                         { Draw curve as solid line }
  DrawAxis(Dx, Dy, X1, Y1, X2, Y2, Lines, Scale, false);
  DrawPolygon(A, 1, N, 7, 2, 0);

  SelectWorld(1);                          { Select outside window }
  SelectWindow(1);

  DrawTextW(730, 450, 1, '^['7@2  The data');  { Print legend }
  DrawTextW(730, 550, 1, '--  The curve');
end; { FindWorldDem }
```

```
begin
  InitGraphic;                          { Initialize the graphics system }

  FindWorldDem;                         { Do the demo }

  repeat until KeyPressed;              { Wait until a key is pressed }

  LeaveGraphic;                         { Leave the graphics system }
end. { FindWorld }
```

The special features of the *DrawAxis* procedure are used to make a border around the drawing, and inset it from the edges of the active window. The inset feature can be used to make room for labels and legends, and to allow multiple drawings in one window.

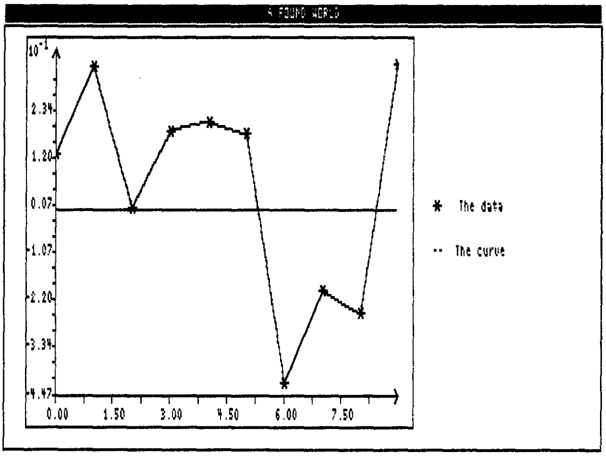


Figure 2-19 *Finding a World for a Polygon (FINDWRLD.PAS Example)*

Solving Curve-Fitting Problems

This section introduces you to the *Spline* and *Bezier* procedures. Both these procedures use polynomials to create curves. However, they are used for different reasons: the *Spline* procedure is used for fitting smooth curves to a given configuration of points, while *Bezier* is used to find the points that will create a desired curve. The *Spline* procedure is appropriate for many curve-fitting applications (for example, creating a smooth curve that intersects a set of experimental data), while *Bezier* is the procedure to use for line modeling and generating curves of arbitrary shape.

The curve produced by the FINDWRLD.PAS example is quite jagged; this is because the data points are connected by straight lines. The *Spline* procedure allows you to take the same set of points and find a smooth curve to fit that configuration of points. The general method used to find the function that will produce such a curve is called *interpolation*; using interpolation, you can generate the "missing" points that will smooth the curve.

The simplest way to interpolate a given set of points with a curve is the following: given *n* points [*X1*,*Y1*],[*X2*,*Y2*],[*X3*,*Y3*]...[*Xn*,*Yn*], we can interpolate the points with the n'th degree polynomial:

$$p_n(x) = y_1 \frac{(x-x_2)\cdots(x-x_n)}{(x_1-x_2)\cdots(x_1-x_n)} + y_2 \frac{(x-x_1)(x-x_3)\cdots(x-x_n)}{(x_2-x_1)(x_2-x_3)\cdots(x_2-x_n)}$$

$$+ \cdots + y_n \frac{(x-x_1)\cdots(x-x_{n-1})}{(x_n-x_1)\cdots(x_n-x_{n-1})}$$

This polynomial is known as the *Lagrange Interpolating Polynomial*, and it generates an exact curve that will pass through all the points. However, there is a problem inherent in this method of interpolation: it requires a formula with the same number of elements as the number of points to be intersected. Interpolating 90 points, for example, will yield a polynomial of degree 90, which is quite unwieldy.

A second, simpler approach to the problem is to make a separate curve in each interval $[x_{i-1},x_i]$, so that the curves meet with no jaggedness or irregularity. In other words, the function consists of pieces of polynomials that are patched together. The method used is known as "*Cubic Splines*". Using this method, 3rd degree polynomials are used in each interval and patched together to form a "smooth" curve.

The Turbo Pascal Graphix *Spline* procedure uses this technique to interpolate the points that make up the curve. To produce the curve, the initial set of points is passed to the *Spline* procedure in the *PlotArray*, along with information about where to start and stop the interpolation, and a second *PlotArray* to receive the points of the smooth curve.

The following example (INTERP.PAS on the distribution disk) is essentially identical to the FINDWRLD.PAS example, except that an additional interpolated curve is plotted. Since the points are plotted at random, running the program several times will give you a good feel for how splines behave.

```
program Interpolate;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow, GShell;

procedure SplineDem;
var
  X, Temp : Float;
  Dx, Dy, I, N, M, Lines, Scale : integer;
  X1, Y1, X2, Y2 : integer;
  B, A : PlotArray;

begin
  DefineWindow(1, 0, 0, XMaxGlb, YMaxGlb); { Define both windows as whole screen }
  DefineWindow(2, 0, 0, XMaxGlb, YMaxGlb);
  DefineWorld(1, 0, 0, 1000, 1000);     { Give a world to the screen }

  DefineHeader(2, 'A spline interpolation'); { Window where curves will go }
  SetHeaderOn;

  N := 12;                              { Fill polygon array }
  for I := 1 to N do
  begin
    A[I, 1] := I - 1;
    A[I, 2] := random - 0.5;
  end;

  M := 50;                              { Generate spline with 50 points }
  Spline(A, N, A[2, 1], A[N - 1, 1], B, M);
  FindWorld(2, B, M, 1, 1.08);          { Make world 2 the right size }

  SelectWindow(2);                      { Select it and draw border }
  DrawBorder;

  Dx := -8;                             { Draw axis inset from window edge }
  Dy := 7;
  X1 := 3;
  Y1 := 5;
  X2 := 25;
  Y2 := 10;
  Lines := 0;
  Scale := 0;

  SetLineStyle(1);                      { Draw initial curve as dotted line }
  DrawAxis(Dx, Dy, X1, Y1, X2, Y2, Lines, Scale, false);
  DrawPolygon(A, 2, N - 1, 7, 2, 0);    { Don't draw the endpoints }

  SetLineStyle(0);                      { Draw interpolated curve as solid line }
  DrawAxis(0, 0, X1, Y1, X2, Y2, 0, 0, false);
  DrawPolygon(B, 1, -M, 0, 0, 0);       { Spline is not good on endpoints }

  SelectWorld(1);                       { Select outside window }
  SelectWindow(1);

  DrawTextW(730, 400, 1, ^['7@2   The data');     { Print legend }
  DrawTextW(730, 500, 1, '..  The initial polygon');
  DrawTextW(730, 600, 1, '__  The interpolated values');
end;  { SplineDem }
```

```
begin
  InitGraphic;                     { Init the graphics system }

  SplineDem;                       { Do the demo }

  repeat until KeyPressed;         { Wait until a key is pressed }

  LeaveGraphic;                    { Leave the graphics system }
end. { Interpolate }
```

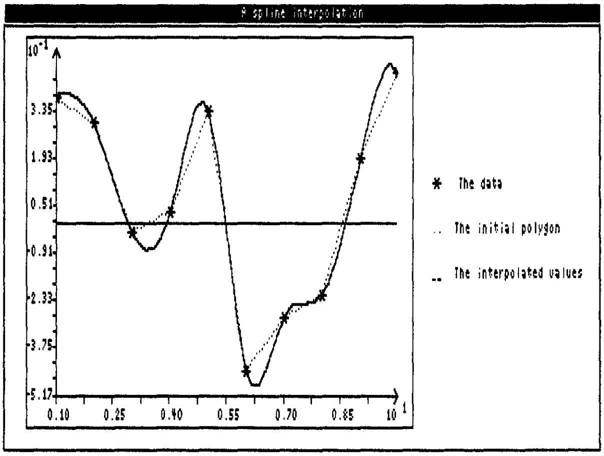


Figure 2-20 Finding a Smooth Curve with Cubic Splines (INTERP.PAS Example)

Modeling a Curve with the Bezier Procedure

The *Bezier* procedure uses polynomials to solve the opposite problem that the *Spline* procedure handles: finding a set of points that will generate a predetermined curve. Bezier polynomials are defined by a given set of guiding (control) points. With the Bezier procedure, you continually redefine these control points so that they "pull on" the curve until it is of the desired shape. Once the guiding points are defined, if you have some talent for mathematics, you can easily find the equations for the corresponding Bezier polynomials that will draw the curve—that is, the algebraic formula for the curve drawn by this procedure. In addition, you can then use these points of the solution to plot the curves as polygons in other windows, using different coordinate systems, or on other screens on different computer systems.

The Bezier polynomial takes the following form:

$$p_x(t) = \sum_{i=0}^{m} C_i^m \; t^i \, (1-t)^{m-i} \, x_i$$

$$p_y(t) = \sum_{i=0}^{m} C_i^m \; t^i \, (1-t)^{m-i} \, y_i$$

where C^i_m is the number of combinations of m objects taken i at a time.

The following example (BEZIDEMO.PAS on the distribution disk) shows you how to use a set of control points to generate a desired curve. This technique is extremely useful for line modeling and some architectural applications. To illustrate the flexibility of the *Bezier* procedure, run this example program and try to make it loop twice.

```
program BeziDemo;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow, GShell;

procedure ClearToEol;
{ Procedure to clear to end of line }
var
  I : integer;
begin
  for I := 1 to 80 do
    Write(' ');
end; { ClearToEol }

procedure ReadInput(var S : WrkString);
const
  Cr = #13;
  Bs = #8;
var
  Count : integer;
  Ch    : char;
begin
  Count := 0;
  S := '';
  repeat
    Ch := ReadKey;
    case Ch of
      Bs : begin
             if Count > 0 then
             begin
               Write(Ch);
               ClrEol;
               Delete(S, Length(S), 1);
               Dec(Count);
             end;
           end;
```

```
    else
      if Ch <> Cr then
      begin
        Write(Ch);
        S := S + Ch;
        Count := Count + 1;
      end;
    end;
  until Ch = Cr;
end; { ReadInput }

procedure BezierDem;

var
  Result, I, MaxControlPoints, MaxIntPoints : integer;
  DummyX, DummyY : Float;
  A, B : PlotArray;
  Break : boolean;
  DummyS, Temp2, Temp : WrkString;

begin
  MaxControlPoints := 7;                { Initialize everything }
  MaxIntPoints := 15;

  A[1, 1] := 1;   A[2, 1] := 1.5; A[3, 1] := 2;   A[4, 1] := 2.5;
  A[5, 1] := 3;   A[6, 1] := 4;   A[7, 1] := 5;   A[1, 2] := 2;
  A[2, 2] := 1.5; A[3, 2] := 1;   A[4, 2] := 2.5; A[5, 2] := 4;
  A[6, 2] := 4.5; A[7, 2] := 5;

  ClearScreen;                          { Set up screen }
  SetColorWhite;
  DefineWorld(1, 0, 0, 6.33, 7.0);      { Set world so rulers are good }
  SelectWorld(1);
  DefineWindow(1, 0, 0, XMaxGlb, 17 * YMaxGlb div 20);
  SelectWindow(1);
  SetBackground(0);
  DrawBorder;
  DrawAxis(7, -7, 0, 0, 0, 0, 0, 0, false);

  Break := false;                       { Init exit flag }

  repeat
    SetLinestyle(1);                    { Draw polygon between points }
    DrawAxis(0, 0, 0, 0, 0, 0, 0, 0, false);
    DrawPolygon(A, 1, MaxControlPoints, 4, 2, 0);

    Bezier(A, MaxControlPoints, B, MaxIntPoints);  { Do bezier operation }

    SetLinestyle(0);                    { Plot it }
    DrawAxis(0, 0, 0, 0, 0, 0, 0, 0, false);
    DrawPolygon(B, 1, MaxIntPoints, 0, 0, 0);

    repeat
      GotoXY(1, 24);                    { Clear out old text }
      ClearToEol;
      GotoXY(1, 25);
      ClearToEol;
      GotoXY(1, 23);
      ClearToEol;
```

```
      GotoXY(1, 23);                   { Get point to change }
      Write('Enter the number of the point to change (0 to quit) :  ');
      GotoXY(55, 23);
      ReadInput(Temp);
      Val(Temp, I, Result);
    until I in [0..MaxControlPoints];

    if I > 0 then
      begin
        repeat
          GotoXY(1, 24);               { Get new values for x and y }
          Write('Old position : [', A[I,1]:4:2, ',', A[I,2]:4:2, ']');
          GotoXY(40, 24);
          Write('   New position  x:  ');
          GotoXY(60, 24);
          ReadInput(DummyS);
          while DummyS[1] = ' ' do
            Delete(DummyS, 1, 1);
          Temp := DummyS;
          GotoXY(40, 25);
          Write('   New position  y:  ');
          GotoXY(60, 25);
          ReadInput(DummyS);
          while DummyS[1] = ' ' do
            Delete(DummyS, 1, 1);
          Temp2 := DummyS;
          Val(Temp, DummyX, Result);
          Val(Temp2, DummyY, Result);
        until ((DummyX >= X1WldGlb) and (DummyX <= X2WldGlb)) and
              ((DummyY >= Y1WldGlb) and (DummyY <= Y2WldGlb));

        SetLinestyle(1);               { Erase old curve }
        SetColorBlack;
        DrawAxis(0, 0, 0, 0, 0, 0, 0, 0, false);
        DrawPolygon(A, 1, MaxControlPoints, 4, 2, 0);
        SetLinestyle(0);
        DrawAxis(0, 0, 0, 0, 0, 0, 0, 0, false);
        DrawPolygon(B, 1, MaxIntPoints, 0, 0, 0);
        A[I, 1] := DummyX;
        A[I, 2] := DummyY;
        SetColorWhite;
      end
    else
      Break := true;                   { Done }
  until Break;
end; { BezierDem }

begin
  InitGraphic;                         { Initialize the graphics system }

  BezierDem;                           { Do the demo }

  LeaveGraphic;                        { Leave the graphics system }
end. { BeziDemo }
```

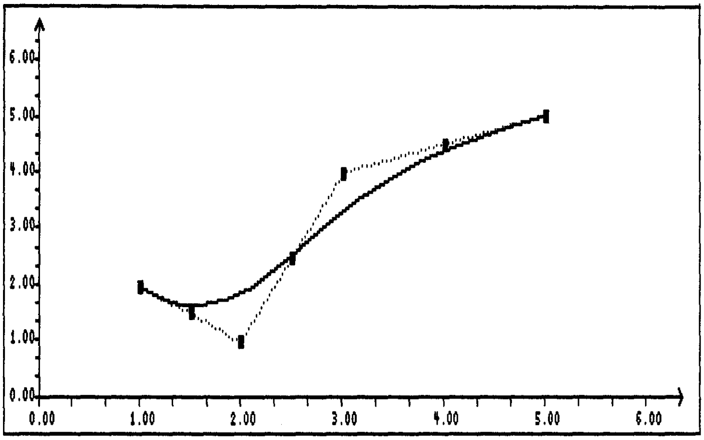


Figure 2-21 *Finding Points to Fit a Smooth Curve of Predetermined Shape (BEZIDEMO.PAS Example)*

Summary of Polygon/Curve Routines

- *Bezier* computes a smooth curve of predetermined shape from a set of control points.
- *DrawAxis* draws *X* and *Y* axes with ruler markings.
- *DrawPolygon* draws a polygon.
- *FindWorld* finds a world coordinate system to fit a given polygon.
- *RotatePolygon* rotates a polygon about its center of gravity.
- *RotatePolygonAbout* rotates a polygon about a given point.
- *Spline* computes a smooth curve from a set of control points.
- *TranslatePolygon* moves a polygon vertically and horizontally.

Screens

There are two types of screens available for drawing with the Turbo Pascal Graphix Toolbox: the displayed screen and a RAM (virtual) screen in memory. Turbo Pascal Graphix routines allow you to save and load either of these screens to and from disk, and restore them when you need them. You can also send images from either screen to your printer, and swap the contents of one screen with the contents of the other.

Use the *SaveScreen* procedure to store the active screen as a file on disk. The single string parameter passed to the routine specifies the file name in which to save the screen contents. If a file with the same name already exists, it is overwritten. When you want to display the screen again, *LoadScreen* retrieves the screen from the file specified by its file name.

Both *SaveScreen* and *LoadScreen* use a format that is screen-type-specific; this means that a screen saved or loaded in a system with one graphics card may not keep its integrity if you attempt to retrieve or save it later on a system with another graphics card. This is also true with the *LoadWindowStack* and *StoreWindowStack* procedures; window stacks are not necessarily compatible between different versions of the Turbo Pascal Graphix Toolbox. However, there is no incompatibility between *individual windows*; you can safely store or load a window using the *LoadWindow* and *StoreWindow* procedures from one graphics screen type to another with no problems.

The following program example (SCREENIO.PAS on the distribution disk) demonstrates saving and loading a screen; included in this example is a routine that draws a Sierpinski curve. This screen image is stored to disk as file DEMO, the screen is cleared, and the image is read back to the screen. *SaveWindow*/*LoadWindow* and *SaveWindowStack*/*LoadWindowStack* are contained in the toolbox unit GWindow. If you use GWindow, you can test this example by substituting their names for the *SaveScreen* and *LoadScreen* procedures, and making sure the data you want to save and load is available.

```
program ScreenIO;

uses
  Dos, Crt, GDriver, Printer, GKernel;

procedure Sierpinski;
const
  N = 5;
var
  I, H, X, Y, X0, Y0 : integer;
  Sec : boolean;

procedure Plot;                         { Draw a line }
begin
  DrawLine(X, Y, X0, Y0);
  X0 := X;
  Y0 := Y;
end;

procedure B(I:integer); forward;        { Forward references for recursion }

procedure C(I:integer); forward;

procedure D(I:integer); forward;
```

```
procedure A(I : integer);              { First recursive procedure }
begin
  if I > 0 then
  begin
    A(I - 1);
    X := X + H;
    Y := Y - H;
    Plot;
    B(I - 1);
    X := X + 2 * H;
    Plot;
    D(I - 1);
    X := X + H;
    Y := Y + H;
    Plot;
    A(I - 1);
  end;
end;  { A }

procedure B;                           { Second recursive procedure }
begin
  if I > 0 then
  begin
    B(I - 1);
    X := X - H;
    Y := Y - H;
    Plot;
    C(I - 1);
    Y := Y - 2 * H;
    Plot;
    A(I - 1);
    X := X + H;
    Y := Y - H;
    Plot;
    B(I - 1);
  end;
end;  { B }

procedure C;                           { Third recursive procedure }
begin
  if I > 0 then
  begin
    C(I - 1);
    X := X - H;
    Y := Y + H;
    Plot;
    D(I - 1);
    X := X - 2 * H;
    Plot;
    B(I - 1);
    X := X - H;
    Y :=Y - H;
    Plot;
    C(I - 1);
  end;
end;  { C }
```

```
procedure D;                        { Last recursive procedure }
begin
  if I > 0 then
  begin
    D(I - 1);
    X := X + H;
    Y := Y + H;
    Plot;
    A(I - 1);
    Y := Y + 2 * H;
    Plot;
    C(I - 1);
    X := X - H;
    Y := Y + H;
    Plot;
    D(I - 1);
  end;
end; { D }

procedure DoIt;                     { Sierpinski main procedure }
begin
  I := 3;
  H := 16;
  X0 := 30;
  Y0 := 240;
  repeat
    I := I + 1;
    X0 := X0 - H;
    H := H div 2;
    Y0 := Y0 + H;
    X := X0;
    Y := Y0;
    A(I - 1);
    X := X + H;
    Y := Y - H;
    Plot;
    B(I - 1);
    X := X - H;
    Y := Y - H;
    Plot;
    C(I - 1);
    X := X - H;
    Y := Y + H;
    Plot;
    D(I - 1);
    X := X + H;
    Y := Y + H;
    Plot;
  until I = N;
end; { DoIt }

begin
  SetHeaderOn;
  DefineWorld(1, -3, -3, 258, 258);
  SelectWorld(1);
  SelectWindow(1);
  DrawBorder;
  DoIt;
end; { Sierpinski }
```

```
begin
  InitGraphic;                        { Init the graphics system }

  DefineHeader(1, 'DEMONSTRATE SCREEN SAVE AND READ TO/FROM DISK');

  SetHeaderOn;                        { Give it a header }

  Sierpinski;                         { Do the curve }

  SaveScreen('DEMO.PIC');             { Save the screen to disk }

  ClearScreen;                        { Clear the screen }

  Delay(1000);                        { Delay so that hard disk or RAM disk
                                        users can see the action }

  LoadScreen('DEMO.PIC');             { Retrieve it from disk }


  repeat until KeyPressed;            { Wait until a key is pressed }

  ClearScreen;

  LeaveGraphic;                       { Leave the graphics system }
end. { ScreenIO }
```

Printing Screens

There are two ways to print screen images. You can either use the Turbo Pascal Graphix *HardCopy* procedure, or the existing screen printing facility of your computer.

The *HardCopy* procedure prints screen images on any printer compatible with the Epson MX, RX, or FX series. Depending on the printer used, several width formats are available. These range from 640 points to 1920 points across the page. Since the standard IBM color graphics screen is 640 pixels wide, one screen will exactly fit across the page if the printer is able to print in the lowest resolution mode.

Some printers do not support all the available modes. For instance, the standard IBM, Epson MX-80-compatible printer will only print in the 960-points-per-line mode (mode 1). If you select any other mode for this printer, it will never enter graphics mode and attempt to print the graphics screen in text characters.

Because of the different resolutions that are possible with *HardCopy*, the horizontal-to-vertical proportions (aspect ratio) of some images may be different on the screen than when the images are printed. Experiment with your printer and the resolution modes available to it to find what works best for you.

There is another way to print screen images using an IBM-compatible printer. First, install the graphics print routine that comes with the computer. Usually, this

is done by running the system program GRAPHICS.COM that is on the MS-DOS system disk. Then, when you want to print a screen image, simply press the PrtSc key; on some keyboards, you must also press the Shift key.

There are a couple of advantages to using this program for printing screens. One is that it works on all Epson-like printers, and another is that it prints the image down the page rather than across it. The screen image fills the whole sheet, and the aspect ratio of the image is very close to that of the screen. Since the image is so large, fine details of the drawing look sharp and clear.

The following example program (SCRNPRNT.PAS on the distribution disk) prints out the screen image used in the SCREENIO.PAS example.

```
program ScreenIO;

uses
  Dos, Crt, GDriver, Printer, GKernel;

procedure Sierpinski;
const
  N = 5;
var
  I, H, X, Y, X0, Y0 : integer;
  Sec : boolean;

procedure Plot;                        { Draw a line }
begin
  DrawLine(X, Y, X0, Y0);
  X0 := X;
  Y0 := Y;
end;

procedure B(I:integer); forward;       { Forward references for recursion }

procedure C(I:integer); forward;

procedure D(I:integer); forward;

procedure A(I : integer);              { First recursive procedure }
begin
  if I > 0 then
  begin
    A(I - 1);
    X := X + H;
    Y := Y - H;
    Plot;
    B(I - 1);
    X := X + 2 * H;
    Plot;
    D(I - 1);
    X := X + H;
    Y := Y + H;
    Plot;
    A(I - 1);
  end;
end;  { A }
```

```
procedure B;                          { Second recursive procedure }
begin
  if I > 0 then
  begin
    B(I - 1);
    X := X - H;
    Y := Y - H;
    Plot;
    C(I - 1);
    Y := Y - 2 * H;
    Plot;
    A(I - 1);
    X := X + H;
    Y := Y - H;
    Plot;
    B(I - 1);
  end;
end;  { B }

procedure C;                          { Third recursive procedure }
begin
  if I > 0 then
  begin
    C(I - 1);
    X := X - H;
    Y := Y + H;
    Plot;
    D(I - 1);
    X := X - 2 * H;
    Plot;
    B(I - 1);
    X := X - H;
    Y := Y - H;
    Plot;
    C(I - 1);
  end;
end;  { C }

procedure D;                          { Last recursive procedure }
begin
  if I > 0 then
  begin
    D(I - 1);
    X := X + H;
    Y := Y + H;
    Plot;
    A(I - 1);
    Y := Y + 2 * H;
    Plot;
    C(I - 1);
    X := X - H;
    Y := Y + H;
    Plot;
    D(I - 1);
  end;
end;  { D }
```

```
procedure DoIt;                         { Sierpinski main procedure }
begin
  I := 3;
  H := 16;
  X0 := 30;
  Y0 := 240;
  repeat
    I := I + 1;
    X0 := X0 - H;
    H := H div 2;
    Y0 := Y0 + H;
    X := X0;
    Y := Y0;
    A(I - 1);
    X := X + H;
    Y := Y - H;
    Plot;
    B(I - 1);
    X := X - H;
    Y := Y - H;
    Plot;
    C(I - 1);
    X := X - H;
    Y := Y + H;
    Plot;
    D(I - 1);
    X := X + H;
    Y := Y + H;
    Plot;
  until I = N;
end;  { DoIt }

begin
  SetHeaderOn;
  DefineWorld(1, -3, -3, 258, 258);
  SelectWorld(1);
  SelectWindow(1);
  DrawBorder;
  DoIt;
end;  { Sierpinski }

begin
  InitGraphic;                          { Init the graphics system }

  DefineHeader(1, 'DEMONSTRATE SCREEN PRINTING'); { Give it a header }

  SetHeaderOn;

  Sierpinski;                           { Do the curve }

  HardCopy(false, 1);                   { Print it }

  repeat until KeyPressed;              { Wait until a key is pressed }

  LeaveGraphic;                         { Leave the graphics system }
end. { ScreenIO }
```

C H A P T E R 3

Technical Reference

This chapter provides detailed information about all the routines contained in the Turbo Pascal Graphix Toolbox. The first section gives an overview of the modular units that you'll need to link into your graphics application programs, along with a sample program. The following section defines and describes the constants and types used in the Turbo Pascal Graphix procedures, the third section provides a quick reference guide to Turbo Pascal Graphix routines, and the final section describes all the functions and procedures contained in the package.

Turbo Pascal Graphix Files

Turbo Pascal Graphix Toolbox is supplied on the distribution disk as an assortment of Turbo Pascal units that you will need to "use" in your application program. These files are organized as modules to allow you to choose only the units you need for compilation into your final program.

The Turbo Pascal Graphix distribution disk will contain some files that are specific to your graphics card or computer system. Such files are named by a filename with a .DVR extension. For example, GRAFCGA.DVR is the IBM Color Graphics Adapter device driver. You must copy the GRAFXXX.DVR file written for your hardware (supplied on the distribution disk) onto the GDRIVER.PAS file before you attempt to run a graphix toolbox program. This is done by invoking the Turbo

Pascal Graphix batch program (see "Getting Started" in Chapter 2.) Failure to do so may cause malfunctioning of Turbo Pascal Graphix programs.

Basic System Units

The following files must be used in all Turbo Pascal Graphix applications, since they contain the global variable declarations, drawing primitives, and system routines that are necessary for drawing.

GDRIVER.PAS	Variables and routines for basic drawing, and for loading and storing screens
GKERNEL.PAS	Primitives for control and initialization of the Turbo Pascal Graphix Toolbox

Supplemental System Units

These files are necessary for applications that use windows, text, or error messages. The only unit that you have to use yourself is the GWINDOW.PAS unit; the other files listed here will be used by your application automatically if they are needed.

GWINDOW.PAS	Routines for moving, loading and storing windows
8X8.FON	High-resolution font for IBM
14X9.FON	High-resolution font for Hercules, AT&T, IBM EGA, IBM 3270
4X6.FON	Turbo Pascal Graphix font
ERROR.MSG	Error message text

High-Level Command Unit

The high-level routines are necessary for more complex graphics applications. All of the high-level routines utilize the procedures contained in the basic system units; you must therefore use those units in order to utilize the high-level procedures.

The high-level routines are contained in the unit GSHELL.PAS which includes the following:

- procedures for finding a world coordinate system to fit a polygon
- procedures that draw coordinate axes and labels
- procedures for drawing polygons
- procedures that rotate, scale, and translate polygons
- procedures that do spline smoothing on polygons
- procedures that do Bezier interpolations on polygons
- procedures that fill (hatch) bars in bar charts
- procedures that draw bar charts
- procedures that draw and label circle segments
- procedures that draw and label pie charts

A Sample Turbo Pascal Graphix Toolbox Program

This sample program demonstrates the essential elements of a Turbo Pascal Graphix Toolbox program.

```
program simple;

uses
  Dos, Crt, GDriver, Printer, GKernel;

begin

  InitGraphic;                {initialize the graphics system}
  DrawBorder;                 {draw a border around the drawing}
                              {area of the active window}
                              {(the dimensions of the active window}
                              {default to 640x200 points)}

  DrawLine(10,10,600,180);         {draw a line}
  DrawSquare(10,10,600,180,false); {draw a square}
  DrawLine(-100,-20,750,320);      {draw a line to demonstrate}
                                   {clipping}

  repeat until KeyPressed;      {hold screen until key is pressed}
  LeaveGraphic;                 {leave the graphics system}
end.
```

Constant and Type Definitions

This section defines and describes, in alphabetical order, the constants and types used in Turbo Pascal Graphix Toolbox routines. Each constant or type is first defined, then described in detail as it applies to various procedures and functions. The Turbo Pascal Graphix file that contains the constant or type is given in brackets next to the constant or type name.

To customize your application, you can change some of the constants and types by altering the GDRIVER.PAS file; however, this should be done with great care, and only after you have made certain that you thoroughly understand the Turbo Pascal Graphix Toolbox program. Otherwise, a system crash or other unpredictable disasters could occur.

AspectFactor [GDRIVER.PAS]

Declaration `const AspectFactor = (depends on system);`

Purpose *AspectFactor* is used to adjust the aspect ratio (horizontal-to-vertical ratio) of a circle or ellipse so that a true circle is drawn on a particular physical screen using a particular graphics board. Without this adjustment, a circle may be drawn in a distorted way—too tall or too wide. This is because the horizontal-to-vertical ratio varies on different monitors.

Remarks The graphics system multiplies the aspect ratio for a given circle or ellipse by the value of *AspectFactor* (which varies with the particular hardware screen installed) to create the desired shape. Multiplying *AspectFactor* by a constant creates ellipses with the same width, but with different heights. *AspectFactor* × 1 creates a true circle on any screen, while *AspectFactor* × 2 gives an ellipse that is twice as tall as it was, and *AspectFactor* ÷ 2 gives one that is half as tall as it was. Varying the aspect ratio varies the height of the drawn figure while keeping the width constant. Thus, if three circles are drawn with aspect ratios of *AspectFactor* ÷ 2, *AspectFactor*, and *AspectFactor* × 2, respectively, the three figures will be tangent to each other at their leftmost and rightmost points, but not at their top and bottom points.

Remarks This constant should not be altered, since it is specific to the graphics hardware in your system.

BackgroundArray [GDRIVER.PAS]

Declaration `type BackgroundArray = array [0..7] of byte;`

Purpose *BackgroundArray* is used by the *SetBackground8* procedure to pass the specified 8x8 bit pattern for filling a window background.

CharFile [GDRIVER.PAS]

Declaration `const CharFile: string[StringSizeGlb] = '4x6.font';`

Purpose *CharFile* contains the file name of the 4x6 pixel font file.

Remarks You can change this constant by altering either the TYPEDEF.SYS file or the main program before you call the *InitGraphic* procedure.

HardwareGrafBase [GDRIVER.PAS]

Declaration `const HardwareGrafBase:word = (depends on system);`

Purpose *HardwareGrafBase* defines the hardware segment address of graphics memory for a particular machine or graphics board.

HeaderSizeGlb [GDRIVER.PAS]

Declaration `const HeaderSizeGlb = 10;`

Purpose *HeaderSizeGlb* defines the vertical dimension, in pixels, of window headers. Its value must be greater than or equal to 6.

Remarks The total vertical drawing area available in a given window is reduced by the size of its header.

IVStepGlb [GDRIVER.PAS]

Declaration `const IVStepGlb = (depends on system);`

Purpose *IVStepGlb* specifies the initial value of *VStep*, the step size (increment) by which windows are moved vertically.

Remarks *IVStep* is used by the Turbo Pascal Graphix program to speed the vertical movement of large windows. Its value varies according to the particular hardware installed. See the *MoveVer* and *SetVStep* procedures.

MaxBackground [GDRIVER.PAS]

Declaration `const MaxBackground:word = (depends on system);`

Purpose *MaxBackground* is a value that specifies the maximum number of available background ("black") colors for a particular hardware configuration: 0 or 15 for IBM versions and 0 for Hercules.

Remarks This constant should not be changed, since it is specific to the graphics hardware installed.

MaxForeground [GDRIVER.PAS]

Declaration `const MaxForeground:word = (depends on system);`

Purpose *MaxForeground* is a value that specifies the maximum number of available foreground ("white") drawing colors for a particular hardware configuration: 15 for IBM (except the PCjr version, which allows only black or white) and 0 for Hercules.

Remarks This constant should not be changed, since it is specific to the graphics hardware installed.

It is illegal to set the foreground and background colors to the same value. See the *SetBackgroundColor* and *SetForegroundColor* procedures for more information.

MaxPiesGlb [GDRIVER.PAS]

Declaration `const MaxPiesGlb = 10;`

Purpose *MaxPiesGlb* specifies the maximum number of sections allowed in a pie chart.

MaxPlotGlb [GDRIVER.PAS]

Declaration `const MaxPlotGlb = 100;`

Purpose *MaxPlotGlb* defines the maximum number of points in a *PlotArray*.

Remarks *PlotArray* is used to store the vertices of polygons. *Bezier*, *DrawHistogram*, *DrawPolygon*, *FindWorld*, *RotatePolygon*, *ScalePolygon*, *Spline*, and *TranslatePolygon* make use of the *MaxPlotGlb* constant.

MaxWindowsGlb [GDRIVER.PAS]

Declaration `const MaxWindowsGlb = 16;`

Purpose *MaxWindowsGlb* specifies the maximum number of defined windows.

MaxWorldsGlb [GDRIVER.PAS]

Declaration `const MaxWorldsGlb = 4;`

Purpose *MaxWorldsGlb* specifies the maximum number of world coordinate systems that can be defined.

Remarks Only one world coordinate system can be used at one time.

MinBackground [GDRIVER.PAS]

Declaration `const MinBackground:word = (depends on system);`

Purpose *MinBackground* specifies the minimum value for the background ("black") color for a particular graphics card: 0 for IBM and 0 for Hercules.

Remarks This constant should not be changed, since it is specific to the graphics hardware installed.

MinForeground [GDRIVER.PAS]

Declaration `const MinForeground:word = (depends on system);`

Purpose *MinForeground* specifies the minimum value for the foreground ("white") drawing color for a particular graphics card: 1 for IBM and 1 for Hercules.

Remarks This constant should not be changed, since it is specific to the graphics hardware installed.

PieArray [GDRIVER.PAS]

Declaration `type PieArray = array [1..MaxPiesGlb] of PieType;`

Purpose *PieArray* is used to pass the definition of a pie chart to the *DrawCartPie* and *DrawPolarPie* procedures; each element of the array defines a single section of the pie. The two fields in the array are *Area* (a real number), and *Text* (a string).

Remarks The maximum number of pie sections is determined by the *MaxPiesGlb* constant.

PlotArray [GDRIVER.PAS]

Declaration `type PlotArray = array [1..MaxPlotGlb, 1..2] of Float;`

Purpose *PlotArray* specifies the vertices of a given polygon, and is used to pass polygons to a procedure.

Remarks In the Turbo Pascal Graphix Toolbox, the term *polygon* can mean any ordered collection of points, possibly (but not necessarily) connected by lines. Thus, a sampling of a sine wave can be called a polygon, though a smooth sine wave with an infinite number of points cannot. The data structure simply contains points. Poly[*i*,1] is the *i*'th *X* coordinate, and Poly[*i*,2] is the *i*'th *Y* coordinate. The maximum number of points in a polygon is determined by the constant *MaxPlotGlb*.

PlotArray is used by *Bezier, DrawHistogram, DrawPolygon, FindWorld, RotatePolygon, ScalePolygon, Spline,* and *TranslatePolygon.*

RamScreenGlb [GDRIVER.PAS]

Declaration `const RamScreenGlb:boolean = true;`

Purpose *RamScreenGlb* determines whether or not a RAM (virtual) screen is allocated for drawing.

Remarks A RAM screen takes up a large chunk of memory (as defined by the constant *ScreenSizeGlb*, in bytes) but it enables you to do many things, such as two-screen animation and smooth window movement over a background (see the *MoveWindow* procedure).

Some hardware configurations allocate dedicated memory for RAM screens; in those cases, *RamScreenGlb* will always be TRUE. See Appendix A for further information.

ScreenSizeGlb [GDRIVER.PAS]

Declaration `const ScreenSizeGlb = (depends on system);`

Purpose *ScreenSizeGlb* specifies the size of the screen (in bytes divided by 2) for a particular hardware configuration.

Remarks This constant should not be altered, since it is specific to the size of the physical screen in your computer; any change to this constant may cause a system crash or unnecessary memory allocation.

StringSizeGlb [GDRIVER.PAS]

Declaration `const StringSizeGlb = 80;`

Purpose *StringSizeGlb* specifies the maximum string length of the type *WrkString*.

Remarks This constant is used by any procedure that requires a text string.

WrkString [GDRIVER.PAS]

Declaration `type WrkString = string[StringSizeGlb];`

Purpose *WrkString* is the string type used by Turbo Pascal Graphix procedures that either require string parameters, or use strings internally.

Remarks The *DefineHeader* and *DrawText* procedures use *WrkString* as their principle parameter.

XMaxGlb [GDRIVER.PAS]

Declaration `const XMaxGlb = (depends on system);`

Purpose *XMaxGlb* specifies the width of the screen in bytes, less 1; that is, the maximum value of an *X* (horizontal) window definition coordinate. The maximum screen width is *XMaxGlb* × 8 + 7.

Remarks This constant should not be changed, since it is specific to the particular hardware configuration.

The *DefineWindow* procedure uses *XMaxGlb* to check whether a window is being defined within the physical screen.

XScreenMaxGlb [GDRIVER.PAS]

Declaration `const XScreenMaxGlb = (XMaxGlb*8+7)`

Purpose *XScreenMaxGlb* specifies the maximum width of the screen for a particular hardware configuration.

Remarks This constant should not be changed, since it is specific to the particular hardware configuration.

YMaxGlb [GDRIVER.PAS]

Declaration `const YMaxGlb = (depends on system);`

Purpose *YMaxGlb* specifies the height of the screen in pixels; that is, the maximum value of a Y (vertical) absolute screen coordinate.

Remarks This constant should not be changed, since it is specific to the particular hardware configuration.

The *DefineWindow* procedure uses *YMaxGlb* to check whether a window is being defined within the physical screen.

Quick Reference Guide to Turbo Pascal Graphix Routines

In the following list, the Turbo Pascal Graphix Toolbox routines are grouped by function into six sections: Initialization and Error, Screens, Windows, Color and Drawing, Text, and Internal. Since the list is designed to help you find routines according to their logical use, and since some routines logically relate to more than one function, a few routines appear in more than one section. The declaration for each routine is listed, followed by its page number.

Initialization and Error

procedure	InitGraphic;	156
procedure	EnterGraphic;	141
procedure	LeaveGraphic;	159
procedure	Error(Proc,Code:integer);	142
function	GetErrorCode:byte;	146
procedure	SetBreakOff;	186
procedure	SetBreakOn;	187
procedure	SetMessageOff;	198
procedure	SetMessageOn;	199
procedure	SetVStep(Step:word);	201
function	HardwarePresent:boolean;	154

Screens

procedure	SelectScreen(I:word);	179
procedure	LoadScreen(FileName:WrkString);	160
procedure	ClearScreen;	99
procedure	CopyScreen;	103
procedure	SaveScreen(FileName:WrkString);	175
function	GetScreen:byte;	148
function	GetScreenAspect:Float;	149
procedure	InvertScreen;	157
procedure	SwapScreen;	207
procedure	HardCopy(Inverse:boolean; Mode:byte);	153

Windows

procedure	SetWindowModeOn;	203
procedure	SetWindowModeOff;	202
function	WindowMode:boolean;	213
procedure	DefineWindow(I,XLow,YLow,XHi,YHi:integer);	109
procedure	RedefineWindow(I,XLow,YLow,XHi,YHi:integer);	167
procedure	DefineTextWindow(I,Left,Up,Right,Down,Border:integer);	107
procedure	DefineWorld(I:integer;XLow,YLow,XHi,YHi:Float);	110
procedure	SelectWorld(I:integer);	181
procedure	SelectWindow(I:integer);	180
function	GetWindow:integer;	151
procedure	SetClippingOn;	189
procedure	SetClippingOff;	188
function	Clip(var X1,Y1,X2,Y2:integer):boolean;	101
function	Clipping:boolean;	102
procedure	SetBackground(Pattern:byte);	183
procedure	SetBackground8(Pattern:BackgroundArray);	184
procedure	DefineHeader(I:integer;Hdr:WrkString);	106
procedure	SetHeaderOn;	194

Color and Drawing

Text

Internal

Procedures and Functions

This section defines and describes, in alphabetical order, all the procedures and functions contained in the Turbo Pascal Graphix Toolbox. The call-up for each procedure or function is given, followed by a detailed description of its function. Remarks, restrictions, and examples are given where appropriate, as well as cross-referencing to related procedures and functions. The Turbo Pascal Graphix file that contains the procedure or function is given in brackets next to the name of the procedure or function.

Refer to page 82 for a description of the constants and types used in these procedures and functions.

BaseAddress [GDRIVER.PAS]

Declaration	`function BaseAddress(Y:word):word;`
Usage	BaseAddress(Y);
Parameters	*Y* : a screen line (0...YMaxGlb)
Function	*BaseAddress* calculates the offset of screen line *Y* in memory.
Remarks	This function is for internal use by the graphics system.
Restrictions	None
Example	`I:=BaseAddress(5);`

I is the offset at the start of screen line 5 (the sixth line on the screen).

```
FillChar(Mem[GrafBase:BaseAddress(9)],XMaxGlb,0);
```

sets the 10th screen line to "black."

Bezier [GSHELL.PAS]

Declaration

```
procedure Bezier(A: PlotArray; N: integer;
   var B:PlotArray;M:integer);
```

Usage

Bezier(A,N,B,M);

Parameters

A: array of *X* and *Y* control points

N: number of control points

B: array of resultant Bezier-function base points

M: desired number of base points in resultant Bezier polynomial curve

Function

Bezier computes a Bezier polynomial curve from an array, *A*, that contains *N* control points. The resultant array, *B*, is filled with *M* base points that constitute a parametric curve. The curve passes through the first and last control points, and passes as close as possible to each of the other points.

A Bezier function is defined by a set of control points (*X* and *Y* values). Within this defined interval, the Bezier function calculates the resultant base points.

Remarks

Bezier polynomials are often used when a smooth curve of some particular form is needed. Increasing the value of *M* smooths the curve, but slows down the computing process.

The specific attributes of Bezier functions and their applications in graphic design are discussed in the book, *Principles of Computer Graphics*, by W. Newmann and R. Sproul.

Restrictions

The maximum values for *N* and *M* are determined by the constant *MaxPlotGlb*, specified in the GDRIVER.PAS file. The default value of *MaxPlotGlb* is 100.

See Also

DrawPolygon
RotatePolygon
ScalePolygon
TranslatePolygon

Example

This example, called BEZIDEMO.PAS on the Turbo Pascal Graphix distribution disk, uses seven control points to draw a curve. Fifteen base points (shown as a dotted line) are generated by this procedure. The positions of the points and the value of *M* can be changed interactively.

```
program BeziDemo;

uses
  Dos, Crt, GDriver, Printer, GKernel, GWindow, GShell;

procedure ClearToEol;
{ Procedure to clear to end of line }
var
  I : integer;
begin
  for I := 1 to 80 do
    Write(' ');
end; { ClearToEol }

procedure ReadInput(var S : WrkString);
const
  Cr = #13;
  Bs = #8;
var
  Count : integer;
  Ch    : char;
begin
  Count := 0;
  S := '';
  repeat
    Ch := ReadKey;
    case Ch of
      Bs : begin
             if Count > 0 then
             begin
               Write(Ch);
               ClrEol;
               Delete(S, Length(S), 1);
               Dec(Count);
             end;
           end;
    else
      if Ch <> Cr then
      begin
        Write(Ch);
        S := S + Ch;
        Count := Count + 1;
      end;
    end;
  until Ch = Cr;
end; { ReadInput }

procedure BezierDem;

var
  Result, I, MaxControlPoints, MaxIntPoints : integer;
  DummyX, DummyY : Float;
  A, B : PlotArray;
  Break : boolean;
  DummyS, Temp2, Temp : WrkString;
```

```
begin
  MaxControlPoints := 7;              { Initialize everything }
  MaxIntPoints := 15;

  A[1, 1] := 1;   A[2, 1] := 1.5; A[3, 1] := 2;   A[4, 1] := 2.5;
  A[5, 1] := 3;   A[6, 1] := 4;   A[7, 1] := 5;   A[1, 2] := 2;
  A[2, 2] := 1.5; A[3, 2] := 1;   A[4, 2] := 2.5; A[5, 2] := 4;
  A[6, 2] := 4.5; A[7, 2] := 5;

  ClearScreen;                        { Set up screen }
  SetColorWhite;
  DefineWorld(1, 0, 0, 6.33, 7.0);  { Set world so rulers are good }
  SelectWorld(1);
  DefineWindow(1, 0, 0, XMaxGlb, 17 * YMaxGlb div 20);
  SelectWindow(1);
  SetBackground(0);
  DrawBorder;
  DrawAxis(7, -7, 0, 0, 0, 0, 0, 0, false);

  Break := false;                     { Init exit flag }

  repeat
    SetLinestyle(1);                { Draw polygon between points }
    DrawAxis(0, 0, 0, 0, 0, 0, 0, 0, false);
    DrawPolygon(A, 1, MaxControlPoints, 4, 2, 0);

    Bezier(A, MaxControlPoints, B, MaxIntPoints);  { Do bezier }
                                                  { operation }

    SetLinestyle(0);                  { Plot it }
    DrawAxis(0, 0, 0, 0, 0, 0, 0, 0, false);
    DrawPolygon(B, 1, MaxIntPoints, 0, 0, 0);

    repeat
      GotoXY(1, 24);                  { Clear out old text }
      ClearToEol;
      GotoXY(1, 25);
      ClearToEol;
      GotoXY(1, 23);
      ClearToEol;
      GotoXY(1, 23);                  { Get point to change }
      Write('Enter the number of the point to change (0 to quit) : ');
      GotoXY(55, 23);
      ReadInput(Temp);
      Val(Temp, I, Result);
    until I in [0..MaxControlPoints];

    if I > 0 then
      begin
        repeat
          GotoXY(1, 24);            { Get new values for x and y }
          Write('Old position : [', A[I,1]:4:2, ',', A[I,2]:4:2, ']');
          GotoXY(40, 24);
          Write('   New position  x:  ');
          GotoXY(60, 24);
          ReadInput(DummyS);
```

```
          while DummyS[1] = ' ' do
            Delete(DummyS, 1, 1);
          Temp := DummyS;
          GotoXY(40, 25);
          Write('   New position  y:  ');
          GotoXY(60, 25);
          ReadInput(DummyS);
          while DummyS[1] = ' ' do
            Delete(DummyS, 1, 1);
          Temp2 := DummyS;
          Val(Temp, DummyX, Result);
          Val(Temp2, DummyY, Result);
        until ((DummyX >= X1WldGlb) and (DummyX <= X2WldGlb)) and
              ((DummyY >= Y1WldGlb) and (DummyY <= Y2WldGlb));

        SetLinestyle(1);                { Erase old curve }
        SetColorBlack;
        DrawAxis(0, 0, 0, 0, 0, 0, 0, 0, false);
        DrawPolygon(A, 1, MaxControlPoints, 4, 2, 0);
        SetLinestyle(0);
        DrawAxis(0, 0, 0, 0, 0, 0, 0, 0, false);
        DrawPolygon(B, 1, MaxIntPoints, 0, 0, 0);
        A[I, 1] := DummyX;
        A[I, 2] := DummyY;
        SetColorWhite;
      end
    else
      Break := true;              { Done }
  until Break;
end; { BezierDem }

begin
  InitGraphic;                    { Initialize the graphics system }

  BezierDem;                      { Do the demo }

  LeaveGraphic;                   { Leave the graphics system }
end. { BeziDemo }
```

ClearScreen [GDRIVER.PAS]

Declaration `procedure ClearScreen;`

Usage ClearScreen;

Function *ClearScreen* erases the screen that is currently in use (the active screen).

Remarks Initialization is not performed by this procedure; see *InitGraphic*.

Restrictions None

See Also InitGraphic

Example

```
program ClearScreenExample;

uses
  Dos, Crt, GDriver, Printer, GKernel;

begin
  DrawLine(1,1,200,200);
  DrawLine(1,200,200,1);
  gotoxy(50,12);
  write('Hit return to clear screen: ');
  readln;
  ClearScreen;
  gotoxy(10,25);
  write('Hit return to end: ');
  readln;
end.
```

ClearWindowStack [GWINDOW.PAS]

Declaration	`procedure ClearWindowStack(Nr:integer);`
Usage	ClearWindowStack(Nr);
Parameters	*Nr*: index of window to be erased [1..*MaxWindowsGlb*]
Function	*ClearWindowStack* deletes a designated window, *Nr*, from the window stack. If there is no window entry at the given index, the operation is not performed.
Remarks	A call to *RestoreWindow* cannot restore a window erased using this routine.
Restrictions	The value of *Nr* must lie between 1 and the constant *MaxWindowsGlb* (defined in the GDRIVER.PAS file).
See Also	ResetWindowStack RestoreWindow StoreWindow
Example	`ClearWindowStack(7);`

removes the window stack entry (if there is one) for window 7.

Clip [GKERNEL.PAS]

Declaration `function Clip(var X1,Y1,X2,Y2:integer):boolean;`

Usage Clip(X1,Y1,X2,Y2);

Parameters *X1,Y1* : coordinates of starting point of line

X2,Y2 : coordinates of end point of line

boolean : if FALSE, line lies outside window

Function *Clip* clips a line to fit the active window, and determines whether or not the full length of a line is drawn. The four integer variables represent absolute screen coordinates. Clip adjusts them as follows: if a line is drawn from [*X1,Y1*] to [*X2,Y2*], any part of the line that lies outside the active window is removed. The resulting coordinates describe a line that is entirely contained by the active window. The boolean function value is TRUE if the adjusted coordinates still represent a line, and FALSE if the entire line is clipped away.

Remarks Although this function is mainly for internal use, it can also be useful when you are working with window mode off (*SetWindowModeOff*), to ensure that drawings remain within the physical screen.

Restrictions Since the four integer parameters are modified by *Clip*, they must be variables; they cannot be expressions.

See Also Clipping
SetClippingOff
SetClippingOn
SetWindowModeOff
SetWindowModeOn

Example

```
if Clip(X1,Y1,X2,Y2) then DrawLine(X1,Y1,X2,Y2);
```

draws only the part of the line that falls within the active window.

```
B:=Clip(X1,Y1,X2,Y2);
```

adjusts [X1,Y1] and [X2,Y2] so that the line between them is entirely contained by the active window; sets B to TRUE if any part of the original line remains.

Clipping [GKERNEL.PAS]

Declaration `function Clipping:boolean;`

Usage Clipping;

Function *Clipping* returns the clipping status: TRUE when clipping is enabled with the *SetClippingOn* procedure; FALSE when clipping is disabled with the *SetClippingOff* procedure.

Restrictions None

See Also Clip
SetClippingOn
SetClippingOff

Example
```
B:=Clipping;
```

sets *B* to TRUE if clipping is enabled, FALSE if not.

CopyScreen [GDRIVER.PAS]

Declaration	`procedure CopyScreen;`
Usage	CopyScreen;
Function	*CopyScreen* copies the active screen onto the inactive screen.
Remarks	If the active screen is the RAM screen, this procedure copies it to the displayed screen. *CopyScreen* is often used to save a window background when another window is being moved over the background. See Chapter 2, page 36 for detailed information about moving windows.
Restrictions	In order to use this procedure, there must be an available RAM screen in memory, that is, the constant *RamScreenGlb* must be TRUE in the GDRIVER.PAS file.
See Also	LoadScreen SaveScreen SelectScreen SetBackground SwapScreen
Example	`CopyScreen;`

copies the active screen onto the inactive screen.

CopyWindow [GWINDOW.PAS]

Declaration `procedure CopyWindow(From,To:byte; X1,Y1:integer);`

Usage CopyWindow(From,To,X1,Y1);

Parameters *From* : screen from which window is copied
To : screen window is to be copied onto
X1, Y1 : window definition coordinates where window is copied

Function *CopyWindow* copies the contents of the active window to and from the RAM screen and the displayed screen. A value of 1 for *To* or *From* designates the displayed screen, while a value of 2 for *To* or *From* designates the RAM screen. The window is copied to the screen location specified by window definition coordinates [*X1,Y1*].

Remarks *CopyWindow* copies images from the area enclosed by the active window in the specified screen. This may have surprising results if the wrong screen is specified!

See page 36 for complete information about moving windows.

Restrictions To use *CopyWindow*, there must be an available RAM screen in memory, that is, the constant *RamScreenGlb* is TRUE (defined in the GDRIVER.PAS file).

See Also LoadWindow
SelectWindow

Example

```
CopyWindow(1,2,10,20);
```

copies the active window from the displayed screen to the RAM screen, placing the upper left corner of the window at window definition coordinates [10,20] (screen coordinates [80,20]).

```
CopyWindow(1,1,50,5);
```

copies the active window from its current position on the displayed screen to window definition coordinates [50,5] (screen coordinates [400,5]) on the displayed screen.

DC [GDRIVER.PAS]

Declaration	`procedure DC(C:byte);`
Usage	DC(C);
Parameter	*C*: ASCII code of drawn character
Function	*DC* draws the character whose ASCII code is *C* at text coordinates [*XTextGlb,YTextGlb*] (internal variables) in the font used by the particular hardware configuration installed.
Remarks	*DC* is for internal use by the graphics system. It does not advance the cursor.
Restrictions	None
See Also	DefineTextWindow TextDown TextLeft TextRight TextUp
Example	`DC(32);` displays character 32 (space) at the current cursor position on the active screen, without moving the cursor.

DefineHeader [GKERNEL.PAS]

Declaration `procedure DefineHeader(I:integer; Hdr:WrkString);`

Usage DefineHeader (I,Hdr);

Parameters *I* : index of window for which header is defined [1..*MaxWindowsGlb*]

Hdr : string term for window header

Function *DefineHeader* defines a window header, *Hdr*, for a given window, *I*. The procedure defines the text that makes up the header, but has no effect on the display; the header is not displayed or altered until procedure *DrawBorder* is called. The header is then centered horizontally either on the top or the bottom of the window, depending on whether the last call was to *SetHeaderToTop* or *SetHeaderToBottom*.

Restrictions Window headers can only be drawn with the 4x6-pixel character set.

The value of *I* must lie between 1 and the constant *MaxWindowsGlb* (defined in GDRIVER.PAS file).

See Also DrawBorder
RemoveHeader
SetHeaderOff
SetHeaderOn
SetHeaderToBottom
SetHeaderToTop

Example

```
DefineHeader(1,'*** Edit window ***');
```

defines the header of window 1 to be *** *Edit window* ***, without affecting the display of the header.

DefineTextWindow [GKERNEL.PAS]

Declaration
```
procedure DefineTextWindow(I,Left,Up,Right,Down,
      Border:integer);
```

Usage DefineTextWindow(I,Left,Up,Right,Down,Border);

Parameters
I : index of window

Left : X coordinate of left edge of machine-dependent text

Up : Y coordinate of upper edge of machine-dependent text

Right : X coordinate of right edge of machine-dependent text

Down : Y coordinate of bottom edge of machine-dependent text

Border : desired number of pixels between text and window boundaries

Function *DefineTextWindow* uses the given text coordinates (*Left, Up, Right, Down, and Border*) and the number of pixels, *Border,* that you want between the text and all four edges of the window, to define a window. The window defined will allow for a uniform space between the text and the window edges.

Remarks *DefineTextWindow* is used to fit and align text within a window. It is particularly useful with the Hercules version of the Turbo Pascal Graphix Toolbox, since Hercules text is defined on 9-pixel boundaries, while windows are defined on 8-pixel boundaries; this one-pixel offset can create alignment problems.

If you wish to vary the space between your text and any of the four window edges, use the *TextLeft, TextRight, TextUp,* and *TextDown* functions to define the space individually for each window edge.

Restrictions If you define a 4x6 pixel header for your window, the placement of the machine-dependent text will be thrown off by the size of the header; in this case, use the four functions mentioned to realign text within the window.

Note that the horizontal border values are only approximate, since they are restricted to window defintion coordinates, and are adjusted outward if necessary.

See Also
DefineHeader
DefineWindow
TextDown
TextLeft
TextRight
TextUp

Example

```
DefineTextWindow(3,2,2,79,24,4);
```

defines window 3 so that it encloses text coordinates from [2,2] to [79,24], with a border of at least 4 pixels between the text and all edges.

DefineWindow [GKERNEL.PAS]

Declaration `procedure DefineWindow(I,XLow,YLow,XHi,YHi:integer);`

Usage DefineWindow(I,XLow,XHi,YHi);

Parameters *I* : index of window [1..*MaxWindowsGlb*]

XLow : *X* value of upper left window position [0..*XMaxGlb*]

YLow : *Y* value of upper left window position [0..*YMaxGlb*]

XHi : *X* value of lower right window position [1..*XMaxGlb*]

YHi : *Y* value of lower right window position [0..*YMaxGlb*]

Function *DefineWindow* defines a region of the screen as a window, *I*. The window is defined as a rectangle with the upper left corner at [*XLow*,*YLow*] and the lower right corner at [*XHi*,*YHi*].

Remarks The *X* coordinates of a window are defined in 8-pixel chunks; that is, windows are placed on byte boundaries in memory. If *DefineWindow* is called with parameters (1,10,10,19,19), the defined window is 10 pixels tall and 80 pixels wide.

Restrictions The value of *I* must be between 1 and *MaxWindowsGlb* (as defined in the GDRIVER.PAS file), all coordinates must lie within the physical screen, and the *Low* coordinates must be lower in numeric value than the *Hi* coordinates; otherwise, an error will occur.

See Also RedefineWindow
SelectWindow

Example

```
DefineWindow(4,5,5,10,10);
```

defines window 4, with upper left corner at window definition coordinates [5,5] and lower right corner at [10,10] (screen coordinates [40,5] and [87,10]).

```
DefineWindow(2,0,0,XMaxGlb div 2,YMaxGlb div 2);
```

defines window 2 as the upper left quarter of the screen.

DefineWorld [GKERNEL.PAS]

Declaration	`procedure DefineWorld(I:integer; XLow,YLow,XHi,YHi:Float);`
Usage	DefineWorld(I,XLow,YLow,XHi,YHi);
Parameters	*I* : index of world to be defined [1...*MaxWorldsGlb*]
	XLow : *X* coordinate of upper left vertex
	YLow : *Y* coordinate of upper left vertex
	XHi : *X* coordinate of lower right vertex
	YHi : *Y* coordinate of lower right vertex
Function	*DefineWorld* defines a world coordinate system, delineated by the rectangle formed by the vertices [*XLow*,*YLow*] and [*XHi*,*YHi*]. World coordinates therefore range from [*XLow*,*YLow*] to [*XHi*,*YHi*].
Remarks	The world coordinate system is not enabled until *SelectWorld* is called.
Restrictions	The world's index value, *I*, must lie between 1 and *MaxWorldsGlb* (as defined in the GDRIVER.PAS file), and the *Low* coordinates must be lower in numeric value than their respective *Hi* coordinates; otherwise, an error will occur.
See Also	DefineWindow SelectWindow SelectWorld
Example	`DefineWorld(1,0,-1,2*Pi,1);` defines a world suitable for displaying one cycle of the sine function.

DP [GDRIVER.PAS]

Declaration `procedure DP(X,Y:word);`

Usage DP(X,Y);

Parameters *X,Y*: coordinates of drawn point

Function *DP* draws a point at screen coordinates [*X,Y*].

Remarks This procedure is primarily for internal use of the graphics system.

Restrictions Since no clipping is performed by this procedure, it is important to specify valid *X* and *Y* parameters; otherwise, program memory may be encroached upon, or the system may crash.

See Also DrawPoint

Example

```
DP(2,3);
```

draws a point at screen coordinates [2,3] on the active screen in the current drawing color.

DrawAscii [GKERNEL.PAS]

Declaration `procedure DrawAscii(var X,Y:integer; Size,Ch:byte);`

Usage DrawAscii(X,Y,Size,Ch);

Parameters *X,Y* : coordinates of drawn character

Size : size of character

Ch : ASCII value of character

Function *DrawAscii* draws a single character with ASCII value *Ch* at screen coordinates [*X,Y*]. The 4x6-pixel character set is used. The character is drawn with its upper left corner at screen coordinates ($X, Y - (2 \times Size) + 1$). Each point of the character is drawn as a *Size*-by-*Size* box, so the character is multiplied by *Size* in both directions. *X* is changed to $X + (6 \times Size)$, so that another call to *DrawAscii* using the same *X* and *Y* variables would draw the next character one position to the right (with a $2 \times Size$ blank space between the characters).

Remarks The character is clipped at the boundaries of the active window if clipping is enabled with *SetClippingOn*. The character would be displayed to the right, and both above and below coordinates [*X,Y*].

Restrictions None

See Also DefineHeader
DrawText
DrawTextW

Example

```
DrawAscii(20,40,25,ord('W'));
```

draws a very large (100x150 pixel) *W* at screen coordinates [20,40]. Modifies *X* so that if another character of that size were drawn, it would be placed directly after the first character.

DrawAxis [GSHELL.PAS]

Declaration

```
procedure DrawAxis(XDensity,YDensity,Left,Top,Right,Bottom:
        integer;XAxis,YAxis:integer;Arrows:boolean);
```

Usage

DrawAxis(XDensity,YDensity,Left,Top,Right,Bottom,XAxis,
YAxis,Arrows);

Parameters

XDensity : density of tick marks on *X* ruler (− 9 to 9)

YDensity : density of tick marks on *Y* ruler (− 9 to 9)

Left : distance of drawing area from left edge of window

Top : distance of drawing area from top edge of window

Right : distance of drawing area from right edge of window

Bottom : distance of drawing area from bottom edge of window

XAxis : line style of horizontal axis

YAxis : line style of vertical axis

Arrows : if TRUE, arrow symbols drawn at ends of axes; if FALSE, arrows not drawn

Function

DrawAxis draws *X* and *Y* axes with ruler markings in the active window, to provide coordinate reference information for plots and drawings. This procedure can optionally define the world drawing area to be smaller than a window, draw a line around the drawing area, provide automatically labeled rulers for *X* and *Y* axes with variable tick mark density, and coordinate axes in various line styles.

The parameters *Left, Top, Right,* and *Bottom* move the drawing area in from the edges of the active window. If these parameters are all equal to 0, the drawing area is the entire window. *XDensity* and *YDensity* select how close together tick marks are drawn on the rulers, from −9 to 9. The sign of the *Density* parameters is ignored, except that if one of the *Density* parameters is negative and the other positive, a line is drawn around the drawing area. The *XAxis* and *YAxis* parameters specify the line styles of the horizontal and vertical axes. If either is negative in value, the corresponding axis is not drawn. The line styles correspond to those used to select line styles in the *DrawLine* procedure.

Restrictions

Moving the drawing area in from the edges of the active window is subject to the following conditions:

1. It only affects procedures *DrawHistogram* and *DrawPolygon*.
2. It is disabled after one call to either *DrawHistogram* or *DrawPolygon*.
3. A new procedure called *ResetAxis* has been added to the AXIS.HGH module. *ResetAxis* sets the global variable *AxisGlb* to TRUE. A typical calling sequence for plotting several curves on a common axis frame would now be:

In GSHELL.PAS

```
DrawAxis(...);       { Define axis frame. }
DrawPolygon(...);    { Plot first curve.  }
ResetAxis;           { Reset the axis.    }
DrawPolygon(...);    { Plot second curve. }
ResetAxis;           { Reset the axis.    }
DrawHistogram(...);  { Plot third curve.  }
```

Example

```
DrawAxis(2,2,0,0,0,0,0,0,false);
```

draws solid axes that extend to the edges of the active window, with arrows on their ends. Numbers on the axes are displayed very far apart.

```
DrawAxis(9,-1,1,4,1,4,1,-1,true);
```

draws a dashed horizontal axis with an arrow on the end, and with numbers displayed very close together. Axis is drawn in an area that is smaller than the active window by 8 pixels on the right and left and 4 pixels on the top and bottom. A border is drawn around the drawing area.

DrawBorder [GKERNEL.PAS]

Declaration	`procedure DrawBorder;`
Usage	DrawBorder;
Function	*DrawBorder* draws a border around the active window in the current drawing color and line style.
Remarks	If a header has been defined for the active window with the *DefineHeader* procedure, *DrawBorder* positions the header on the upper edge of the window if *SetHeaderToTop* has been called, or on the lower edge of the window if *SetHeaderToBottom* has been called. A header reduces the available drawing area in the window; if no header is defined, the whole window is used as the drawing area. *DrawBorder* does not erase the active window. If you need to erase the window background, use *SetBackground* (set to 0).
Restrictions	If the header is too long to fit within the window, it is not drawn. (Header length $\times$ 6) must be less than the width of the window in pixels, -2.
See Also	DefineHeader DrawSquare SetBackground SetHeaderOff SetHeaderOn SetHeaderToBottom SetHeaderToTop
Example	`DrawBorder;` draws a border around the active window, along with a header or footer if one was previously defined with the *DefineHeader* procedure.

DrawCartPie [GSHELL.PAS]

Declaration

```
procedure DrawCartPie(XCenter,YCenter,XStart,YStart,
      Inner,Outer:Float;A:PieArray; N,Option,
      Scale:integer;
```

Usage

DrawCartPie(XCenter,YCenter,XStart,YStart,Inner,Outer, A,N,Option,Scale);

Parameters

XCenter,YCenter : world coordinates of center point of circle

XStart,YStart : world coordinates of starting point of first circle segment

Inner : inner radius of label line in radius units

Outer : outer radius of label line in radius units

A : pie chart array

N : number of circle segments

Option : labeling options
Option = 0 : no label
Option = 1 : text label only
Option = 2 : text and numeric label
Option = 3 : numeric label only

Scale : multiplier for specifying size of label

Data Format

Pie chart data is passed to the procedure as an array of the following form:

```
type PieType=record
                Area:Float;
                Text:wrkstring;
     end;
PieArray=array [1..MaxPiesGlb] of PieType;
```

Function

DrawCartPie draws a pie chart, referenced to the *X* and *Y* coordinates of the starting point of the first pie segment, with optional text or numeric labels. Each segment's area and label are passed to the procedure in the *PieArray, A*, which defines the pie chart to be drawn.

DrawCartPie first determines each segment's proportion of the whole pie chart, then draws and labels the segments. Each segment's percentage of the pie chart is determined by totaling the areas of all segments, then displaying each segment's area as a percentage of the total area. Since this computation of percentage is not affected by the absolute values of the areas, any number

system can be used for specifying the areas. A negative value for area causes the pie segment to move out radially and be displayed separately from the rest of the pie chart.

A line is drawn from each pie segment, starting at a distance of *Inner* away from the center segment and ending at a distance of *Outer*. A text and/or numeric label can be drawn at the end of each segment line in the 4x6-pixel character set. *Inner* and *Outer* specify the inner and outer radii that the radial label line is to traverse, with 1 being on the circle itself. *Option* specifies whether the area value and/or text is displayed; a value of 0 designates no label, 1 specifies text label only, 2, text and numeric label, and 3, numeric label only. *Scale* specifies the size of the characters that make up the label.

Remarks

Pie segments are drawn in a clockwise direction. Any part of the pie chart that lies outside the window boundaries is clipped if clipping is enabled with the *SetClippingOn* procedure.

Note that the aspect ratio is applied to pie charts. The aspect ratio must be set to 1 with the *SetAspect* procedure to ensure a circular pie chart.

To draw a pie chart with reference to its radius and the angle of its first segment, use *DrawPolarPi segment, use DrawPolarPie*.

See Also

DrawCircleSegment
DrawPolarPie
PieArray (type)
SetAspect

Example

```
DrawCartPie(100,100,125,100,1.1,1.4,SalesFigures,9,2,1);
```

draws a pie chart, with 9 sections, from the *SalesFigures* array. The starting point of the first pie segment is at [125,100]. Both numeric and text labels are attached to the pie with short lines. Labels are drawn in 4x6-pixel characters without scaling.

DrawCircle [GKERNEL.PAS]

Declaration `procedure DrawCircle(X,Y,R:Float);`

Usage DrawCircle(X,Y,R);

Parameters *X,Y* : coordinates of point at center of circle or ellipse

R : radius of circle or ellipse

Function *DrawCircle* draws circles and ellipses. The circle or ellipse is drawn with its radius measured in the horizontal (*X*) direction, and with *Radius* × *Aspect* in the vertical (*Y*) direction.

Remarks The horizontal-to-vertical ratio (aspect ratio) is set with the procedure *SetAspect*. Small aspects produce ellipses stretched horizontally, and large aspects produce vertical ellipses, while an aspect of 1 draws a true circle.

Restrictions If *SetWindowModeOn* has been called, the value of the radius must be divided by 100, that is, a radius value of 0.1 gives a circle with a radius of 10.

If *SetWindowModeOff* has been called, then the radius value you specify is the radius you get, that is, a radius value of 10 gives a circle with radius 10.

See Also AspectFactor (constant)
DrawCircleDirect
DrawCircleSegment
SetAspect

Example `DrawCircle(20,40,15)`

draws a circle whose center point is at coordinates [20,40] with a radius of 15.

DrawCircleDirect [GKERNEL.PAS]

Declaration `procedure DrawCircleDirect(X,Y,R:integer; Clip:boolean);`

Usage DrawCircleDirect(X,Y,R,Clip);

Parameters *X,Y* : screen coordinates of point at center of circle or ellipse

R : radius of circle or ellipse

Clip : enables/disables clipping

Function *DrawCircleDirect* draws a circle or ellipse, with the radius measured in *X* units of the screen. If *Clip* is TRUE, the circle is clipped at window boundaries; if FALSE, the circle is not clipped.

Remarks This procedure is used for fast circle drawing. It should be used with caution, since it could cause drawing outside the physical screen. *DrawCircle* should be used in applications where speed of operations is not crucial.

Restrictions None

See Also DrawCircle
SetAspect

Example `DrawCircleDirect(100,100,100,true);`

draws a circle at screen coordinates [100,100] with a radius of 100 pixels, without clipping at window boundaries.
WITH

DrawCircleSegment [GSHELL.PAS]

Declaration

```
procedure DrawCircleSegment(XCenter,YCenter:Float;var XStart,
          YStart:Float;Inner,Outer,Angle,Area:Float;Text:
          WrkString;Option,Scale:byte);
```

Usage

DrawCircleSegment(XCenter,YCenter,XStart,YStart,Inner,Outer, Angle,Area,Text,Option,Scale);

Parameters

XCenter, YCenter	:	coordinates of point at center of circle
XStart, YStart	:	coordinates of starting point of segment
Inner	:	inner radius of label line in radius units
Outer	:	outer radius of label line in radius units
Angle	:	angle of segment in degrees
Area	:	numeric label corresponding to segment
Text	:	text label corresponding to segment
Option	:	display options
Option = 0	:	no label
Option = 1	:	text label only
Option = 2	:	text and numeric label
Option = 3	:	numeric label only
Scale	:	multiplier used to determine the size of label

Function

DrawCircleSegment draws an arc of a circle with optional text and numeric labels. The center of the circle is at coordinates [*XCenter, YCenter*] (world coordinates), and the starting point of the arc is at coordinates [*XStart, YStart*]. The angle of the arc is passed directly in degrees. A line segment pointing outwards from the arc is drawn starting at a distance *Inner* away from the arc, and continuing to a distance *Outer*. After the segment is drawn, the coordinates of the endpoint are passed back through the starting position variables.

Text and/or numeric labels can be added. A radial label line can be drawn from the center of the circle segment outward to any location; its inside starting point is specified by *Inner* and its outside radius by *Outer*. *Inner* and *Outer* are scaled radius values: a value of 1 specifies a point on the circle segment, 0.5 a point halfway between the circle segment and its center, and a value of 2 indicates a point one radius distance outside the circle segment. A value of 1 for both inner and outer radii effectively disables the line so it does not appear. The outer radius determines where the label is to be placed. The *Option* parameter specifies whether to type text and/or

numerics as the label; a value of 0 specifies no label, 1 specifies text label only, and 2, both text and numeric label. Labels are drawn in the 4x6-pixel character set. *Scale* determines the size of the characters in the label.

Remarks

If part of the segment lies outside the defined window boundaries and *SetClippingOn* has been called, the segment is clipped at window boundaries.

The aspect ratio is used by this procedure; see the *SetAspect* procedure.

Restrictions

If *Inner* or *Outer* is equal to 0, the label line is not drawn.

See Also

AspectFactor (constant)
DrawCartPie
DrawCircle
DrawCircleDirect
DrawPolarPie
SetAspect

Example

```
DrawCircleSegment(X,Y,ArcX,ArcY,1.1,1.4,30,2300,'Capital
      gains: $',2,1)
```

draws an arc starting at [*ArcX,ArcY*] that extends 30 degrees counterclockwise, centered around coordinates [*X,Y*]. A line is added with label saying *Capital gains: $2300* in 4x6-pixel characters.

DrawCross [GKERNEL.PAS]

Declaration `procedure DrawCross(X,Y,Scale:integer);`

Usage DrawCross(X,Y,Scale);

Parameters *X,Y* : coordinates of point at center of cross

Scale : multiplier for specifying size of cross

Function *DrawCross* draws a cross (+) at coordinates [*X*,*Y*]. The size of the cross is approximately 2**Scale* × 2**Scale*.

Remarks This procedure is primarily for internal use of the graphics system; it is used by *DrawPolygon* to mark lines.

Restrictions None

See Also DrawCrossDiag
DrawDiamond
DrawStar
DrawWye

Example `DrawCross(137,42,5);`

draws a cross at screen coordinates [137,42].

DrawCrossDiag [GKERNEL.PAS]

Declaration `procedure DrawCrossDiag(X,Y,Scale:integer);`

Usage DrawCrossDiag(X,Y,Scale);

Parameters *X,Y* : coordinates of point at center of cross

Scale : multiplier for specifying size of cross

Function *DrawCrossDiag* draws a diagonal cross (x) at coordinates [*X*,*Y*]. The size of the diagonal cross is approximately 2**Scale* × 2**Scale*.

Remarks This procedure is primarily for internal use of the graphics system; it is used by *DrawPolygon* to mark lines.

Restrictions None

See Also DrawCross
DrawDiamond
DrawStar
DrawWye

Example `DrawCrossDiag(89,70,8);`

draws a diagonal cross at screen coordinates [89,70].

DrawDiamond [GKERNEL.PAS]

Declaration	`procedure DrawDiamond(X,Y,Scale:integer);`
Usage	DrawDiamond(X,Y,Scale);
Parameters	*X,Y* : coordinates of point at center of diamond
	Scale : multiplier for specifying size of diamond
Function	*DrawDiamond* draws a diamond (◇) at coordinates [*X,Y*]. The size of the diamond is approximately 2**Scale* × 2**Scale*.
Remarks	This procedure is primarily for internal use of the graphics system; it is used by *DrawPolygon* to mark lines.
Restrictions	None
Example	`DrawDiamond(470,40,4);`
	draws a diamond at screen coordinates [470,40].

DrawHistogram [GSHELL.PAS]

Declaration
```
procedure DrawHistogram(A:PlotArray; N:integer;
                        Hatching:boolean; HatchStyle:integer);
```

Usage DrawHistogram(A,N,Hatching,HatchStyle);

Parameters

A : array of bar chart

N : number of bars in chart

Hatching : enable or disable hatching

HatchStyle : density of hatching
negative value = positive slope direction
positive value = negative slope direction

Data Format Bar chart data is passed in an array of the type *PlotArray,* with the following form:

$A[i,1]$ = not used
$A[i,2]$ = height of the i'th bar (Y value)

Function *DrawHistogram* can create many types of bar charts with different hatchings and an optional axis display.

DrawHistogram draws a bar chart from an array, *A*, of real number values, [*MaxPlotGlb*,2]. *DrawHistogram* uses the [*i*,2] elements of the array to determine the height of each bar. The array is somewhat compatible with a polygon array, in that the *Y* axis components are displayed with constant increments in the *X* dimension. *DrawHistogram* calculates these increments from the window display width and the number of elements in the array to be displayed. The height of the histogram bars are scaled using the world coordinate system active at the time. The bars can be displayed in two modes: they can either be drawn from the bottom of the display area, or from the *Y* axis. When *N*, which specifies the number of bars in the chart, is positive, the bars are drawn from the bottom of the display area, and the absolute value function is applied. This forces all values to be positive, and thus prevents negative values from overwriting the ruler display when it is near the *Y* axis. When *N* is negative, bars are drawn from the *Y* axis, and the actual positive and negative values are used.

If *Hatching* is TRUE, each bar is hatched. The density and direction of the hatch lines is determined by *HatchStyle*. The value of *HatchStyle* determines the number of pixels between hatch lines; a value of 1 gives solid bars with no hatching, with increasing values widening the space between bars. The sign of the

HatchStyle value determines the initial direction of hatching; hatching direction alternates with each consecutive bar. If *HatchStyle* is negative, the initial hatch line is drawn with a positive slope; if *HatchStyle* is positive, it is drawn with negative slope.

Remarks — The active window is entirely filled horizontally with the bar chart.

Restrictions — The number of bars is limited by the constant *MaxPlotGlb*, as defined in the GDRIVER.PAS file.

See Also — MaxPlotGlb (constant)

Example

```
DrawHistogram(BarChartPoints,-40,true,6);
```

draws a bar chart with 40 bars in the active window. The bars may go up or down from the (invisible) horizontal axis, and they are hatched sparsely.

DrawLine [GKERNEL.PAS]

Declaration	`procedure DrawLine(X1,Y1,X2,Y2:Float);`
Usage	DrawLine(X1,Y1,X2,Y2);
Parameters	*X1,Y1* : coordinates of starting point of line
	X2,Y2 : coordinates of end point of line
Function	*DrawLine* draws a line from [*X1,Y1*] to [*X2,Y2*] in the line style selected by the *SetLinestyle* procedure.
Remarks	The line is drawn in world coordinates unless the window mode is disabled with the *SetWindowModeOff* procedure, in which case the line is drawn in absolute screen coordinates. With window mode enabled, any part of the line that lies outside the window boundaries is clipped.
See Also	DrawStraight SetLinestyle
Restrictions	None
Example	`DrawLine(40,107.5,99,50)`

draws a line between world coordinates [40,107.5] and [99,50].

DrawLineClipped [GKERNEL.PAS]

Declaration	`procedure DrawLineClipped(X1,Y1,X2,Y2:integer);`
Usage	DrawLineClipped(X1,Y1,X2,Y2);
Parameters	*X1,Y1* : coordinates of starting point of line *X2,Y2* : coordinates of end point of line
Function	*DrawLineClipped* is a special procedure used to draw a line safely when the window mode is disabled with the *SetWindowModeOff* procedure. The line is drawn in absolute screen coordinates.
Remarks	This procedure clips a line at the active window boundaries, regardless of whether window mode is on or off.
Restrictions	None
See Also	DrawLine
Example	`DrawLineClipped(1,1,199,199);` draws a line between screen coordinates [1,1] and [199,199].

DrawPoint [GKERNEL.PAS]

Declaration `procedure DrawPoint(X,Y:Float);`

Usage DrawPoint(X,Y);

Parameters X,Y : coordinates of point

Function *DrawPoint* draws or redraws a point at coordinates [*X*,*Y*]. If window mode is enabled with the *SetWindowModeOn* procedure, the point is drawn in the active window in world coordinates and is scaled accordingly; if window mode is disabled with the *SetWindowModeOff* procedure, the point is drawn in absolute screen coordinates.

Remarks If clipping is enabled with the *SetClippingOn* procedure, the point is clipped (not displayed) if it lies outside the active window boundaries.

See Also SetColorBlack
SetColorWhite

Restrictions None

Example

```
DrawPoint(35.9,50.2)
```

draws a point at world coordinates [35.9,50.2].

```
For Phi:=0 To 359 Do
  DrawPoint(Phi,Sin(Phi*Pi/180));
```

draws one cycle of a sine wave, with the world coordinate system defined by coordinates [0, – 1] and [359,1].

DrawPolarPie [GSHELL.PAS]

Declaration

```
procedure DrawPolarPie(XCenter,YCenter,Radius,Angle,
          Inner,Outer:Float;A:PieArray; N,Option,
          Scale:integer;
```

Usage

DrawPolarPie(XCenter,YCenter,Radius,Angle,Inner,Outer,
A,N,Option,Scale);

Parameters

XCenter,YCenter	:	world coordinates of center point of circle
Radius	:	radius of pie
Angle	:	angle of first pie segment (in degrees)
Inner	:	inner radius of label line in radius units
Outer	:	outer radius of label line in radius units
A	:	pie chart array
N	:	number of circle segments
Option	:	labeling options
Option = 0	:	no label
Option = 1	:	text label only
Option = 2	:	text and numeric label
Option = 3	:	numeric label only
Scale	:	multiplier for specifying size of label

Data Format

Pie chart data is passed to the procedure as an array of the following form:

```
type PieType=record
               Area:Float;
               Text:wrkstring;
             end;
     PieArray=array [1..MaxPiesGlb] of PieType;
```

Function

DrawPolarPie draws a pie chart, referenced to its radius and the angle of its first segment, with optional text or numeric labels. Each segment's area and label are passed to the procedure in the *PieArray, A*, which defines the pie chart to be drawn.

DrawPolarPie first determines each segment's proportion of the whole pie chart, then draws and labels the segments. Each segment's percentage of the pie chart is determined by totaling the areas of all segments, then displaying each segment's area as a percentage of the total area. Since this computation of percentage is not affected by the absolute values of the areas, any number system can be used for specifying the areas. A negative value for

area causes the pie segment to move out radially and be displayed separately from the rest of the pie chart.

A line is drawn from each pie segment, starting at a distance of *Inner* away from the center segment and ending at a distance of *Outer*. A text and/or numeric label can be drawn at the end of each segment line in the 4x6-pixel character set. *Inner* and *Outer* specify the inner and outer radii that the radial label line is to traverse, with 1 being on the circle itself. *Option* specifies whether the area value and/or text is displayed; a value of 0 designates no label, 1 specifies text label only, and 2, text and numeric label. *Scale* specifies the size of the characters that make up the label.

Remarks

Pie segments are drawn in a clockwise direction. Any part of the pie chart that lies outside the window boundaries is clipped if clipping is enabled with the *SetClippingOn* procedure.

Note that the aspect ratio is applied to pie charts. The aspect ratio must be set to 1 with the *SetAspect* procedure to ensure a circular pie chart.

To draw a pie chart in reference to the starting point (*X* and *Y* coordinates) of its first segment, use *DrawCartPie*.

See Also

DrawCartPie
DrawCircleSegment
PieArray (type)
SetAspect

Example

```
DrawPolarPie(100,100,50,45,1.1,1.4,SalesFigures,9,2,1);
```

draws a pie chart, with 9 sections, from the *SalesFigures* array. Its radius is 50, and its first segment has a 45 degree angle. Both numeric and text labels are attached to the pie with short lines. Labels are drawn in 4x6-pixel characters without scaling.

DrawPolygon [GSHELL.PAS]

Declaration

```
procedure DrawPolygon(A:PlotArray; First,Last,Code,Scale,
                      Lines:integer);
```

Usage

DrawPolygon(A,First,Last,Code,Scale,Lines);

Parameters

A : polygon vertex array (see data format)

First : array index of first vertex to plot

Last : array index of last vertex to plot

Code : code of a graphic symbol

Scale : multiplier for specifying size of symbol (scaling)

Lines : choice of bar presentation

Data Format

The coordinates of the points of a polygon are passed in the global array *PlotArray*. The data type *PlotArray* is defined as follows:

```
type PlotArray = array [1..MaxPlotGlb,1..2] of Float;
```

MaxPlotGlb is a constant that gives the maximum number of vertices (points) of a polygon. This number is preset to 100, but may be changed to any number by editing the GDRIVER.PAS file.

The coordinates of the points must be presented in the following manner:

$A[i,1]$ = X coordinate of the i'th point
$A[i,2]$ = Y coordinate of the i'th point

Function

DrawPolygon draws a polygon using line segments with variable attributes and vertex symbols. The polygon is drawn in the active window in the current drawing color and line style.

First and *Last* define the range of the array, *A*. When *Last* is negative, all drawings are clipped. This is useful after rotations, moves, scaling, or after the world coordinate system has been changed. When *Last* is positive, only symbols are clipped, and drawing takes places faster than in the full clipping mode.

First specifies the array index of the first vertex to plot. If any of the following conditions are not fulfilled, an error occurs.

$First < \text{abs}(Last)$
$First > 0$
$\text{abs}(Last) - First \geq 2$

Code specifies the code for a graphics symbol. If *Code* is a negative value, only symbols are displayed at vertices; if it is positive, sym-

bols are displayed at vertices, and the vertices are connected with lines in the line style chosen by the *SetLinestyle* procedure. The following list shows the available symbols, along with their codes.

Code	**Symbol**
0	line
1	(+)
2	(×)
3	(□)
4	(■)
5	(◇)
6	(Y)
7	(*)
8	(O)
9	(.)
>9	line

Scale determines the size of the symbol; its value must always be greater than 1.

Line determines whether or not vertical lines are drawn from the axis to the vertices. Options are:

Line < 0 : lines are drawn from Y-zero-axis to each vertex

Line = 0 : no lines

Line > 0 : lines are drawn up from bottom of display area to each vertex

Remarks — To draw coordinate axes for the polygon, call *DrawAxis* before calling *DrawPolygon*.

Restrictions — None

See Also — PlotArray (type)

Example — `DrawPolygon(Points,10,30,8,1,0);`

draws the 10th through 30th points of the *Points* array. Points are displayed as small circles, and are connected by lines drawn in the current line style.

DrawSquare [GKERNEL.PAS]

Declaration `procedure DrawSquare(X1,Y1,X2,Y2:Float; Fill:boolean);`

Usage DrawSquare(X1,Y1,X2,Y2,Fill);

Parameters *X1,Y1* : world coordinates of point at upper left corner of rectangle

X2,Y2 : world coordinates of point at lower right corner of rectangle

Fill : enables/disables filling of rectangle

Function *DrawSquare* draws a rectangle, with point [*X1,Y1*] as the upper left corner and point [*X2,Y2*] as the lower right corner. The rectangle is drawn in the line style selected by the *SetLineStyle* procedure. When *Fill* is TRUE, the rectangle is filled with the current drawing color.

Remarks To fill a square with a background pattern, define the square as a window and use *SetBackground* or *SetBackground8*.

Restrictions None

See Also DrawLine
DrawStraight
SetForegroundColor
SetLinestyle

Example `DrawSquare(2,3,50,90,true);`

draws a solid rectangle defined by world coordinates [2,3] through [50,90].

DrawSquareC [GKERNEL.PAS]

Declaration **procedure** `DrawSquareC(X1,Y1,X2,Y2:integer; Fill:boolean);`

Parameters

X1,Y1 : screen coordinates of point at upper left corner of rectangle

X2,Y2 : screen coordinates of point at lower right corner of rectangle

Fill : enables/disables filling of rectangle

Function *DrawSquareC* draws a rectangle, with coordinate [*X1,Y1*] at the upper left corner and coordinate [*X2,Y2*] at the lower right corner of the rectangle. The rectangle is drawn in screen coordinates, but is clipped at the boundaries of the active window.

Remarks This procedure is used internally by the *DrawBorder* procedure.

Restrictions None

See Also DrawSquare

Example

```
DrawSquareC(2,3,50,90,false);
```

draws the part of the square (defined by screen coordinates [2, 3] and [50, 90]) that fits in the active window.

DrawStar [GKERNEL.PAS]

Declaration `procedure DrawStar(X,Y,Scale:integer);`

Usage DrawStar(X,Y,Scale);

Parameters *X,Y* : coordinates of center point of star

Scale : multiplier for determining size of star

Function *DrawStar* draws a six-pointed star (*) at coordinates [*X,Y*]. The size of the star is approximately 2**Scale* × 2**Scale* (in pixels).

Remarks This procedure is mainly for internal use by the graphics system; it is used by *DrawPolygon* for marking lines.

Restrictions None

See Also DrawCross
DrawCrossDiag
DrawDiamond
DrawWye

Example `DrawStar(400,130,30);`

draws a large star at screen coordinates [400, 130].

DrawStraight [GDRIVER.PAS]

Declaration	`procedure DrawStraight(X1,X2,Y:word);`
Usage	DrawStraight(X1,X2,Y);
Parameters	*X1* : *X* screen coordinate of starting point of line *X2* : *X* screen coordinate of end point of line Y : *Y* screen coordinate of line
Function	*DrawStraight* draws a horizontal line from [*X1*,*Y*] to [*X2*,*Y*] in absolute screen coordinates; no clipping is performed.
Remarks	Although *DrawLine* can accomplish the same function as *DrawStraight*, this procedure performs the task of drawing horizontal lines much faster. *DrawStraight* is useful for speedy filling of squares. The line is always drawn in line style 0 (See *SetLinestyle*).
Restrictions	None
See Also	DrawLine SetLinestyle
Example	`DrawStraight(23,502,100);` draws a long horizontal line between screen coordinates [23,100] and [502,100].

DrawText [GKERNEL.PAS]

Declaration	`procedure DrawText(X,Y,Scale:integer; Text:WrkString);`
Usage	DrawText(X,Y,Scale,Text);
Parameters	*X,Y* : coordinates of point at beginning of character string
	Scale : multiplier for specifying character size
	Text : character string

Function

DrawText draws the given string, *Text*, beginning at screen coordinates [*X,Y*]. The procedure uses the 4x6-pixel character set multiplied both vertically and horizontally by *Scale*. If an ESCape (character 27 decimal) is in the string, a particular symbol is drawn according to the next character in the string.

There are eight possible symbols, corresponding to the sequences ESC 1 through ESC 8:

1 = +
2 = ×
3 = □
4 = ■
5 = ◇
6 = 'Y'
7 = *
8 = O

The symbols are drawn to the same scale as the text. The ESCape sequence can also be given in the form ESC *n* @ *s*, where *n* is a number between 1 and 8 and *s* is an integer value. In this case, ESC *n* designates which symbol to draw, while *s* specifies the scale of the symbol. For instance, the sequence ESC 1 @ 5 would draw a cross with a scale of 5.

Remarks

Text is clipped at active window boundaries if *SetClippingOn* has been called.

Restrictions

None

See Also

DrawTextW

Example

```
DrawText(100,100,2,'Some text');
```

draws the character string *Some text* beginning at screen coordinates [100,100], in 8x12-pixel characters.

```
DrawText(250,19,3,'This is a diamond: '+Chr(27)+'5';
```

draws the character string *This is a diamond* ◇ beginning at screen coordinates [250, 19] in 12x18-pixel characters.

DrawTextW [GKERNEL.PAS]

Declaration `procedure DrawTextW(X,Y,Scale:Float; Text:WrkString);`

Usage DrawTextW(X,Y,Scale,Text);

Parameters *X,Y* : world coordinates of point at beginning of character string

Scale : multiplier for specifying character size

Text : character string

Function *DrawTextW* draws the given string, beginning at world coordinates [*X*,*Y*]. The procedure uses the 4x6-pixel character set multiplied both vertically and horizontally by *Scale*. If an ESCape (character 27 decimal) is in the string, a particular symbol is drawn according to the next character in the string.

Remarks See *DrawText* for possible ESC sequence symbols.

Restrictions None

See Also DisplayChar
DrawText

Example

```
DrawTextW(5.7,19.02,3,'This text starts at
      (5.7,19.02) in world coordinates');
```

draws the character string beginning at world coordinates (5.7,19.02) in the active window.

DrawWye [GKERNEL.PAS]

Declaration	`procedure DrawWye(X,Y,Scale:integer);`
Usage	DrawWye(X,Y,Scale);
Parameters	*X,Y* : coordinates of center point of *Y* symbol *Scale* : multiplier for specifying size of symbol
Function	*DrawWye* draws a Y-shaped symbol at coordinates [*X,Y*]. The size of the *Y* is approximately 2**Scale* × 2**Scale*.
Remarks	This procedure is mainly for internal use by the graphics system; it is used by *DrawPolygon* for marking lines.
Restrictions	None
See Also	DrawCross DrawCrossDiag DrawDiamond DrawStar
Example	`DrawWye(50,90,4);`

draws a *Y*-shaped figure at screen coordinates [50, 90].

EnterGraphic [GDRIVER.PAS]

Declaration	`procedure EnterGraphic;`
Usage	EnterGraphic;
Function	*EnterGraphic* turns the graphics mode on and clears the screen. This procedure is normally called to reactivate the graphics mode after *LeaveGraphic* has been called.
Remarks	*EnterGraphic* does not initialize the graphics system; to do that, *InitGraphic* must be called. *EnterGraphic* also loads the system-dependent (higher quality) character set the first time it is called, and sets *ConOutPtr* to point to *DisplayChar*. After *EnterGraphic* is called, "black" will be true black and "white" will be true white, regardless of the graphics card installed. A call to *SetBackgroundColor* or *SetForegroundColor*, followed by a call to *EnterGraphic*, will cancel the colors set by the *SetColor* procedures and set them to true black and white.
Restrictions	None
See Also	InitGraphic LeaveGraphic
Example	`EnterGraphic;` clears the screen, sets the colors to true black and white, and turns graphics mode on.

Error [GKERNEL.PAS]

Declaration `procedure Error(Proc,Code:integer);`

Usage Error(Proc,Code);

Parameters *Proc* : address of procedure where error was detected

Code : error code

Function *Error* is called when an error is discovered by one of the Turbo Graphix procedures; the address of the procedure and an error code are given. If break mode is enabled with the *SetBreakOn* procedure, NO an error will halt the program and an error message and a trace of the addresses of the procedures in the calling sequence that caused the error are displayed. If break mode is disabled with the *SetBreakOff* procedure, the *Error* procedure stores the error code for later examination when the *GetErrorCode* function is called. If message mode is enabled with the *SetMessageOn* procedure, a message is displayed, regardless of *SetBreakOn/Off*.

Remarks Error messages are displayed on line 25 of the screen. If *SetBreakOn* has been called, a list of addresses is displayed. The first address given is the location of the call to *Error*. If the compiler *Option Find* is used on the program, the compiler will display the statement *Error(p,c)* for the graphics procedure where the error was detected. The next number is the address of the statement that called the procedure that found the error. Each successive number is the caller of the previous procedure. The last number points out the line in the main program that started the fatal calling sequence.

Restrictions None

See Also GetErrorCode
SetBreakOff
SetBreakOn
SetMessageOff
SetMessageOn

Example `Error(2,3);`

signals error code 3 in procedure 2.

FindWorld [GSHELL.PAS]

Declaration	`procedure FindWorld(I:integer; A:PlotArray; N:integer;` `ScaleX,ScaleY:Float);`
Usage	FindWorld (I,A,N,ScaleX,ScaleY);
Parameters	*I* : index of world *A* : polygon array *N* : number of vertices in polygon array *ScaleX* : additional scaling factor in *X* direction *ScaleY* : additional scaling factor in *Y* direction

Function

FindWorld determines a world coordinate system for a polygon. The procedure finds the maximum and minimum *X* and *Y* values used to draw a polygon, and then defines a world that either exactly encloses the polygon, or that is larger or smaller by some chosen percentage (*ScaleX* and *ScaleY*). *FindWorld* automatically executes *DefineWorld* and *SelectWorld* procedures after it determines the appropriate world coordinate system.

The *I* parameter selects the index of the world that is to be modified. The selected polygon, *A*, is passed in the *PlotArray*, with *N* specifying the number of vertices in the polygon. The *X* and *Y* dimensions can be multipled by *ScaleX* and *ScaleY*, respectively, to adjust the scaling of the world dimensions; this allows extra space around the polygon or changes its proportions. If no extra scaling is desired, *ScaleX* and *ScaleY* should be set to 1.

Remarks

For a more complete description of the data structure for polygons of the type *PlotArray*, refer to the description for *DrawPolygon*.

Restrictions

None

See Also

DefineWorld
DrawPolygon
PlotArray (type)
SelectWorld

Example

```
FindWorld(1,Diagram,30,1,2);
```

sets a world coordinate system 1, so that the 30-point polygon *Diagram* exactly fits the world horizontally, and half fills it vertically.

GetAspect [GKERNEL.PAS]

Declaration	`function GetAspect:Float;`
Usage	GetAspect;
Function	*GetAspect* returns the current value of the aspect ratio.
Remarks	See *SetAspect* for complete information on the aspect ratio.
See Also	AspectFactor (constant) GetScreenAspect SetAspect SetScreenAspect
Restrictions	None
Example	`R:=GetAspect;`

R gets the current aspect ratio.

GetColor [GKERNEL.PAS]

Declaration	`function GetColor:word;`
Usage	GetColor;
Function	*GetColor* returns the drawing color: 0 if the current color is "black", and 255 if it is "white".
Remarks	"Black" and "white" can be any color available to the particular graphics card installed. For more information on hardware configuration, see Chapter 1 and Appendix A.
See Also	SetBackgroundColor SetColorBlack SetColorWhite SetForegroundColor
Restrictions	None
Example	`I:=GetColor;` *I* is 0 if the current drawing color is black, or 255 if the current drawing color is white.

GetErrorCode [GKERNEL.PAS]

Declaration `function GetErrorCode:byte;` integer

Usage GetErrorCode;

Function *GetErrorCode* returns the error code of the most recent error, or −1 if no error occurred. A call to *GetErrorCode* resets the error code to −1.

The error codes are

−1 : No error
0 : Error msg missing
1 : Font file missing
2 : Index out of range
3 : Coordinates out of range
4 : Too few array elements
5 : Error opening file
6 : Out of window memory
7 : Value(s) out of range

Remarks *GetErrorCode* should be called after the use of any routine that could cause the errors listed here.

Restrictions The procedure or function that caused the error cannot be determined with this routine; see *SetBreakOn*.

See Also Error
SetBreakOff
SetBreakOn
SetMessageOff
SetMessageOn

Example
```
If GetErrorCode < >-1 Then ShutDown;
```

executes procedure *ShutDown* if any graphics error has occurred.

GetLineStyle [GKERNEL.PAS]

Declaration `function GetLineStyle:word;`

Usage GetLineStyle;

Function *GetLineStyle* returns the current line style (selected by *SetLine-Style*), an integer from 0 to 4, or 256 to 511.

Restrictions None

See Also SetLineStyle

Example `I:=GetLineStyle;`

I gets a value in the ranges 0..4 and 256..511, representing the current line style.

GetScreen [GKERNEL.PAS]

Declaration `function GetScreen:byte;`

Usage GetScreen;

Function *GetScreen* returns the code corresponding to the RAM (virtual) or displayed screen currently in use (active): code 1 if the displayed screen is active, or 2 if the RAM screen is active.

Restrictions None

See Also SelectScreen

Example `I:=GetScreen;`

I is 1 if the displayed screen is the active screen, or 2 if the RAM screen is active.

GetScreenAspect [GKERNEL.PAS]

Declaration	`function GetScreenAspect:Float;`
Usage	GetScreenAspect;
Function	*GetScreenAspect* returns the current pixel value of the aspect ratio.
Remarks	See *SetAspect* for complete information on the aspect ratio.
See Also	AspectFactor (constant) GetAspect SetAspect SetScreenAspect
Restrictions	None
Example	`R:=GetScreenAspect;`

R gets the current aspect ratio, in pixels.

GetVStep [GKERNEL.PAS]

Declaration `function GetVStep:word;`

Usage GetVStep;

Function *GetVStep* returns the current value of the step (single increment) for vertical window movement.

Remarks See *SetVStep* for explanation of *step*.

Restrictions None

See Also SetVStep

Example `I:=GetVStep;`

I is the current vertical step value.

GetWindow [GKERNEL.PAS]

Declaration	`function GetWindow:integer;`
Usage	GetWindow;
Function	*GetWindow* returns the code number of the active window (selected by *SelectWindow*).
Restrictions	None
See Also	SelectWindow
Example	`I:=GetWindow;`

I is the code number of the active window.

GotoXY [GKERNEL.PAS]

Declaration `procedure GotoXY(X,Y:word);`

Usage GotoXY(X,Y);

Parameters *X,Y* : coordinates of character

Function *GotoXY* positions the text cursor (invisible cursor that determines where next character is to be drawn) at coordinates [*X,Y*].

Remarks This procedure tells *DisplayChar* where to draw the next character, and thereby augments Turbo's normal *GotoXY* procedure.

Restrictions None

See Also DC

Example `GotoXY(1,20);`

causes the cursor to be positioned at the first character on screen line 20.

HardCopy [GKERNEL.PAS]

Declaration `procedure HardCopy(Inverse:boolean;Mode:byte);`

Usage HardCopy(Inverse,Mode);

Parameters *Inverse* : enables/disables reverse video printout

Mode : specifies print mode

Function *HardCopy* supplies a printed copy of the active screen. If *Inverse* is TRUE, the image is printed with black and white reversed. *Mode* specifies the density of the printed image. Seven modes are available:

0,4,5 = 640 points/line (Epson mode 4)

1 = 960 points/line (Epson mode 1)

2 = 960 points/line (Epson mode 2)

3 = 1920 points/line (Epson mode 3)

6 = 720 points/line (Epson mode 6)

Remarks This procedure can be used with Epson printers of series MX, RX, and FX. Pre-FX series printer can be used, but with Mode 1 only. See the Epson printer manuals for more information.

Restrictions Non-Epson printers are not supported.

See Also Epson printer manuals
SelectScreen

Example

```
HardCopy(false,3);
```

causes the active screen to be printed in Epson graphics mode 3.

HardwarePresent [GDRIVER.PAS]

Declaration `function HardwarePresent:boolean;`

Usage HardwarePresent;

Function *HardwarePresent* checks whether or not the necessary graphics hardware is installed in the system (i.e., IBM Color graphics adapter for IBM version, Hercules card for Hercules version, and so on) and returns TRUE if found. If *HardwarePresent* is FALSE, an error occurs.

Remarks This is an internal function called by *InitGraphic*.

This function is useful in a program that uses graphics mode only for certain presentations. If *HardwarePresent* is FALSE, those graphic presentations are not available.

Restrictions If *InitGraphic* is called when *HardwarePresent* is FALSE, the program is terminated.

See Also InitGraphic

Example

```
If Not HardwarePresent Then
 WriteLn('No graphics board detected in your computer.  Make
 another selection');
Else
 Begin
 {Do Graphics}
End;
```

Hatch [GSHELL.PAS]

Declaration `procedure Hatch(X1,Y1,X2,Y2:Float;Delta:integer);`

Usage Hatch(X1,Y1,X2,Y2,Delta);

Parameters *X1,Y1* : coordinates of point at upper left corner of rectangle to be hatched

X2,Y2 : coordinates of point at lower right corner of rectangle to be hatched

Delta : distance between hatch lines

Function *Hatch* shades a rectangular area of the screen defined by world coordinates [*X1,Y1*] and [*X2,Y2*]. The hatch pattern is formed with diagonal lines separated by a distance of *Delta*. A *Delta* value of 1 gives solid hatching (no space between lines), a *Delta* value of 2 gives 50% filled space, a value of 3 gives 33-1/3% filled space, and so on. If *Delta* is positive, the lines are drawn from the upper left to the lower right; if *Delta* is negative, the lines are drawn from the lower left to the upper right.

Remarks If window mode is disabled with the *SetWindowModeOff* procedure, the rectangle is drawn in absolute screen coordinates.

Restrictions None

See Also DrawSquare

Example

```
Hatch(5,5,30,17,4);
```

hatches part of the active window, defined by coordinates [5, 5] and [30, 17], with diagonal lines that fill 1/4 of the given area.

InitGraphic [GKERNEL.PAS]

Declaration `procedure InitGraphic;`

Usage InitGraphic;

Function *InitGraphic* initializes the Turbo Graphix Toolbox. It must be called before any other graphics procedure or function, but may only be called once within a program. *InitGraphic* selects the displayed screen as the active screen and erases it. All windows and worlds are initialized. In addition, *InitGraphic* performs these functions:

- Checks for the presence of appropriate graphics hardware
- Reads in the error messages file
- Reads in the 4x6-pixel character set
- Allocates the RAM screen if *RamScreenGlb* is TRUE in the GDRIVER.PAS file
- Sets aspect ratio to machine-dependent default
- Sets vertical window move step (increment) to machine-dependent default

InitGraphic calls the following procedures:

EnterGraphic;
HardwarePresent;
SelectWindow (1);
SelectWorld (1);
SelectScreen (1);
SetAspect (AspectFactor);
SetBackgroundColor (MinBackground);
SetBreakOn;
SetClippingOn;
SetColorWhite;
SetForegroundColor (MaxForeground);
SetHeaderOff;
SetHeaderToTop;
SetLineStyle (0);
SetMessageOn;
SetWindowModeOn;

Restrictions *InitGraphic* can be called only once within a program.

See Also EnterGraphic
LeaveGraphic

Example `InitGraphic;`

initializes the graphics system and turns on graphics mode.

InvertScreen [GDRIVER.PAS]

Declaration	**procedure** `InvertScreen;`
Usage	InvertScreen;
Function	*InvertScreen* inverts the screen display by changing pixels from black to white or white to black.
Remarks	"Black" and "white" can be any color available to the particular graphics card installed in your system. See Chapter 1 and Appendix A for more information on hardware configuration.
Restrictions	None
See Also	InvertWindow
Example	`InvertScreen;`

changes each pixel on the active screen from "black" to "white", or from "white" to "black."

InvertWindow [GWINDOW.PAS]

Declaration	`procedure InvertWindow;`
Usage	InvertWindow;
Function	*InvertWindow* inverts the active window display by changing pixels from black to white or white to black.
Remarks	"Black" and "white" can be any color available to the particular graphics card installed in your system. See Chapter 1 and Appendix A for more information on hardware configuration.
Restrictions	None
See Also	InvertScreen
Example	`InvertWindow;`

changes each pixel on the active window from "black" to "white," or from "white" to "black."

LeaveGraphic [GDRIVER.PAS]

Declaration `procedure LeaveGraphic;`

Usage LeaveGraphic;

Function *LeaveGraphic* turns the graphics mode off and returns the system to text mode (which was active before *InitGraphic* was called).

Restrictions None

See Also EnterGraphic
InitGraphic

Example `LeaveGraphic;`

turns graphics mode off and text mode on.

LoadScreen [GDRIVER.PAS]

Declaration	`procedure LoadScreen(FileName:WrkString);`
Usage	LoadScreen(FileName);
Parameters	*FileName*: screen file name (as saved on disk)
Function	*LoadScreen* opens the file containing a graphics screen, named *FileName*, and reads the screen onto the active RAM or displayed screen.
Restrictions	Screens saved with one version of the Turbo Pascal Graphix Toolbox are not necessarily compatible with any other version. See Appendix A for more on system compatibility.
See Also	SaveScreen StoreScreen
Example	`LoadScreen('SCREEN.1');`

loads the contents of the file *SCREEN.1* into the active screen.

LoadWindow [GWINDOW.PAS]

Declaration `procedure LoadWindow(I,X,Y:integer;FileName:WrkString);`

Usage LoadWindow(I,X,Y,FileName);

Parameters *I* : index of window to be loaded

X,Y : world coordinates of point where window is loaded

FileName : window file name (as stored on disk)

Function *LoadWindow* loads a window, *I*, named by *FileName*, to position [*X,Y*] (world coordinates). If *X* or *Y* is negative, the previous (saved) *X* or *Y* coordinate value is used (i.e., the negative value is ignored.) The window is loaded into the active window, thereby erasing the contents of the old window.

Remarks *FileName* can include both the filename and an extension, and a disk drive declaration (e.g., *b: filename.xxx*). Windows saved with different versions of the Turbo Pascal Graphix Toolbox are compatible. However, this is not the case for the *LoadScreen* procedure.

Restrictions If a negative value is given for *X* or *Y*, the previous (saved) value for that coordinate is used.

See Also LoadScreen
SaveScreen
SaveWindow

Example

```
LoadWindow(3,-1,20,'WINDOW.3');
```

loads the contents of the file *WINDOW.3* into window 3, using the *X* position previously stored in the file, and the new *Y* position (20).

LoadWindowStack [GWINDOW.PAS]

Declaration `procedure LoadWindowStack(FileName:WrkString);`

Usage LoadWindowStack(FileName);

Parameters *FileName*: filename of window stack (as stored on disk)

Function *LoadWindowStack* stores a window stack, named *FileName*, from disk to window memory. This procedure automatically searches for two files, *FileName.STK* (file containing the stack) and *FileName. PTR* (a pointer file); therefore, you should not add an extension to *FileName*.

Remarks When loading a window stack from a floppy or hard disk, the entire contents of the existing window stack are destroyed.

Restrictions Window stacks saved by different versions of the Turbo Pascal Graphix Toolbox will not necessarily be compatible. See Appendix A for more information on compatibility between systems.

See Also LoadWindow
SaveWindow
SaveWindowStack

Example

```
LoadWindowStack('STACK');
```

loads a window stack from the files *STACK.STK* and *STACK.PTR*.

MoveHor [GWINDOW.PAS]

Declaration `procedure MoveHor(Delta:integer;FillOut:boolean);`

Usage MoveHor(Delta,FillOut);

Parameters *Delta* : distance window is moved

FillOut : enable/disable copy from RAM screen

Function *MoveHor* moves the active window horizontally by *Delta* steps (8 pixels per step). If *FillOut* is FALSE, the area that used to be under the window is filled with the opposite of the current color; if *FillOut* is TRUE, and there is a RAM screen allocated (*RamScreenGlb* is TRUE in GDRIVER.PAS), the area is filled with the corresponding area of the inactive screen. Thus, to move a window over a background, the background must be stored in the inactive screen (with *CopyScreen*) before the window to be moved is drawn on the active screen. The background is then copied from the inactive screen as the window moves.

Restrictions None

See Also MoveVer
SetBackground

Example `MoveHor(-7,false);`

moves the active window by 7 *X* window definition coordinates (56 pixels) to the left, filling the former location of the window with the opposite of the current drawing color.

MoveVer [GWINDOW.PAS]

Declaration `procedure MoveVer(Delta:integer;FillOut:boolean);`

Usage MoveVer(Delta,FillOut);

Parameters *Delta* : distance window is moved

FillOut : enable/disable copy from RAM screen

Function *MoveVer* moves the current window vertically by *Delta* steps (1 pixel per step). If *FillOut* is FALSE, the area that used to be under the window is filled with the opposite of the current color; if *FillOut* is TRUE, and there is a RAM screen allocated (*RamScreenGlb* is TRUE in GDRIVER.PAS file), the area is filled with the corresponding area of the inactive screen. Thus, to move a window over a background, the background must be stored to the inactive screen (with *CopyScreen*) before the window to be moved is drawn on the active screen. The background is then copied from the inactive screen as the window moves.

Remarks *SetVStep* can be called to specify the number of pixels to move a window vertically at one time; this will speed the vertical movement of the window. For example, if *Delta* is 10 and *VStep* is 3, the window will move 3 times by 3, then once by 1, for a total move of 10 pixels (in the time a 4-pixel move would take without the use of *SetVStep*).

Restrictions None

See Also MoveHor

Example `MoveVer(20,true);`

moves the active window 20 pixels toward the bottom of the screen, filling the former location of the window with the contents of the same location on the inactive screen.

PD [GDRIVER.PAS]

Declaration `function PD(X,Y:word):boolean;`

Usage PD(X,Y);

Parameters *X,Y*: screen coordinates of point

Function *PD* checks whether a point has been drawn at screen coordinates [*X,Y*]. *PD* returns TRUE if a point exists at [*X,Y*] in the current drawing color; otherwise, it returns FALSE.

Restrictions None

See Also DrawPoint
PointDrawn

Example `B:=PD(5,5);`

B is TRUE if the point at screen coordinates [5,5] is set to the current drawing color.

```
PointCount:=0
for X:=0 to XScreenMaxGlb do
  for Y:=0 to YMaxGlb do
    if PD(X,Y) then PointCount:=PointCount+1;
```

This program counts the number of points on the screen.

PointDrawn [GKERNEL.PAS]

Declaration `function PointDrawn(X,Y:Float):boolean;`

Usage PointDrawn(X,Y);

Parameters *X,Y*: world coordinates of point

Function *PointDrawn* checks whether or not a point has been drawn at world coordinates [*X,Y*]. *PointDrawn* returns TRUE if a point exists at [*X,Y*] in the current drawing color; otherwise, it returns FALSE.

Restrictions None

See Also DrawPoint
PD

Example `B:=PointDrawn (12.3,17.8)`

B is TRUE if the point at world coordinates [12.3, 17.8] is set in the current drawing color.

RedefineWindow [GKERNEL.PAS]

Declaration `procedure RedefineWindow(I,XLow,YLow,XHi,YHi:integer);`

Usage RedefineWindow(I,XLow,XHi,YHi);

Parameters *I* : index of window [1..*MaxWindowsGlb*]

XLow : *X* value of upper left window position [0..*XMaxGlb*]

YLow : *Y* value of upper left window position [0..*YMaxGlb*]

XHi : *X* value of lower right window position [1..*XMaxGlb*]

YHi : *Y* value of lower right window position [0..*YMaxGlb*]

Function *RedefineWindow* redefines the dimensions of an existing window, *I*. The window is defined as a rectangle with the upper left corner at [*XLow,YLow*] and the lower right corner at [*XHi,YHi*]. The previously defined window header is not affected by *RedefineWindow*.

Remarks The *X* coordinates of a window are defined in 8-pixel chunks; that is, windows are placed on byte boundaries in memory. If *RedefineWindow* is called with parameters (1, 10, 10, 19, 19), the defined window is 10 pixels tall and 80 pixels wide.

Restrictions The value of *I* must be between 1 and *MaxWindowsGlb* (as defined in the GDRIVER.PAS file), all coordinates must lie within the physical screen, and the *Low* coordinates must be lower in numeric value than the *Hi* coordinates; otherwise, an error will occur.

See Also DefineWindow
SelectWindow

Example

```
RedefineWindow(4,5,5,10,10);
```

redefines window 4, with upper left corner at window definition coordinates [5, 5] and lower right corner at [10, 10] (screen coordinates [40, 5] and [87, 10]).

```
RedefineWindow(2,0,0,XMaxGlb div 2,YMaxGlb div 2);
```

redefines window 2 as the upper left quarter of the screen.

RemoveHeader [GKERNEL.PAS]

Declaration	`procedure RemoveHeader(I:integer);`
Usage	RemoveHeader(I);
Parameters	*I* : index for window
Function	*RemoveHeader* removes the header from window *I*. As with *DefineHeader*, this procedure has no effect on the display of the header; the header is erased only when *DrawBorder* is called again.
Remarks	Once the header is removed, the drawing area of the window will include the part of the window that had been occupied by the header.
Restrictions	None
See Also	DefineHeader DrawBorder SetHeaderOn SetHeaderToBottom SetHeaderToTop
Example	`RemoveHeader(8);` removes the header of window 8, so that a subsequent call to *DrawBorder* will not draw the header.

ResetWindows [GKERNEL.PAS]

Declaration	`procedure ResetWindows;`
Usage	ResetWindows;
Function	*ResetWindows* sets all windows to the size of the screen, selects Window 1 (see the *SelectWindow* procedure), and removes all headers. This procedure does not affect the current screen display, but further drawings will be scaled according to absolute screen coordinates.
Remarks	This procedure resets windows in the same way as *InitGraphic*.
Restrictions	None
See Also	InitGraphic ResetWindowStack ResetWorlds SelectWindow
Example	`ResetWindows;`

sets all windows to the size of the screen.

ResetWindowStack [GWINDOW.PAS]

Declaration `procedure ResetWindowStack;`

Usage ResetWindowStack;

Function *ResetWindowStack* erases all windows contained in memory. All windows saved in the window stack are discarded, and all the space allocated for the window stack becomes available for the storage of new windows.

Remarks This procedure initializes the window stack in the same way as *InitGraphic*. It is especially useful for long programs that require several different drawing environments.

Windows saved in the window stack are dynamically allocated with Turbo Pascal's *GetMem* and *FreeMem* procedures. Because of this, the *Mark/Release* method of memory management must not be used.

Restrictions None

See Also InitGraphic
ResetWindows
ResetWorlds
RestoreWindow
StoreWindow

Example

```
ResetWindowStack;
```

discards any windows saved on the window stack.

ResetWorlds [GKERNEL.PAS]

Declaration	`procedure ResetWorlds;`
Usage	ResetWorlds;
Function	*ResetWorlds* sets all worlds to the size of the physical screen and selects World 1. (See the *SelectWorld* procedure.) Further drawings will be scaled to absolute screen coordinates.
Remarks	This procedure resets worlds in the same way as *InitGraphic*.
Restrictions	None
See Also	InitGraphic ResetWindows ResetWindowStack SelectWorld
Example	`ResetWorlds;`

sets all worlds to the size of the screen.

RestoreWindow [GWINDOW.PAS]

Declaration `procedure RestoreWindow(I,DeltaX,DeltaY:integer);`

Usage RestoreWindow(I,DeltaX,DeltaY);

Parameters *I* : index of window [1..*MaxWindowsGlb*]

DeltaX : *X* offset

DeltaY : *Y* offset

Function *RestoreWindow* takes a window, *I*, that was stored in the window stack with the *StoreWindow* procedure and places it on the screen. If *I* is negative, the restored window is then discarded from the window stack. If no saved window is available under the given index number, an error will occur. The *DeltaX* and *DeltaY* parameters give the *X* and *Y* offsets used to position the window on the screen. A value of 0 for both *DeltaX* and *DeltaY* positions the window in the same place it was when it was saved with *StoreWindow*. A value of 1 for both *DeltaX* and *DeltaY* moves the window horizontally by 8 pixels and vertically by 1 pixel.

Restrictions The value of *I* must lie between 1 and the constant *MaxWindowsGlb* (defined in GDRIVER.PAS file).

See Also ClearWindowStack
LoadWindow
StoreWindow

Example

```
RestoreWindow(4,10,0);
```

restores the saved copy of window 4 to the active screen, at its previous *Y* position but 10 *X* window definition coordinates (80 pixels) to the right of its previous *X* position.

RotatePolygon [GSHELL.PAS]

Declaration `procedure RotatePolygon(A:PlotArray; N:integer; Angle:Float);`

Usage RotatePolygon(A,N,Angle);

Parameters
A : polygon array to be rotated
N : number of polygon vertices
Angle : rotation angle in degrees

Function *RotatePolygon* rotates a polygon A, containing *N* vertices, around its center of gravity in a counterclockwise direction by *Angle* degrees. The center of gravity is calculated with the assumption that each vertex has equal weight.

Remarks When displaying a rotated polygon using *DrawPolygon*, the number of vertices should be given as a negative value; this guarantees that the polygon will be clipped at window boundaries.

Use *RotatePolygonAbout* to rotate a polygon about an arbitrary point.

Restrictions None

See Also
DrawPolygon
RotatePolygonAbout
ScalePolygon
TranslatePolygon

Example

```
RotatePolygon(Image,45,37.5);
```

changes the values of the 45 coordinate pairs in *Image* so that the polygon is rotated 37.5 degrees clockwise about its center of gravity.

RotatePolygonAbout [GSHELL.PAS]

Declaration

```
procedure RotatePolygonAbout(A:PlotArray; N:integer;
          Angle,X,Y:Float);
```

Usage RotatePolygonAbout(A,N,Angle,X,Y);

Parameters *A* : polygon array to be rotated

N : number of polygon vertices

Angle : rotation angle in degrees

X,Y : world coordinates of point around which polygon is rotated

Function *RotatePolygonAbout* rotates a polygon *A* containing *N* vertices about an arbitrary point [*X,Y*], in a counterclockwise direction by *Angle* degrees.

Remarks When displaying a rotated polygon with *DrawPolygon*, the number of vertices should be given as a negative value, to ensure clipping at window boundaries.

Restrictions None

See Also DrawPolygon
RotatePolygon
ScalePolygon
TranslatePolygon

Example

```
RotatePolygonAbout(Image,45,37.5,30.5,99);
```

changes the values of the 45 coordinate pairs in *Image* so that the polygon is rotated 37.5 degrees clockwise about world coordinates [30.5, 99].

SaveScreen [GDRIVER.PAS]

Declaration	`procedure SaveScreen(FileName:WrkString);`
Usage	SaveScreen(FileName);
Parameters	*FileName* : file name of screen (as saved on disk)
Function	*SaveScreen* stores a displayed or RAM screen on a floppy or hard disk. If a file with name *FileName* already exists, it is overwritten.
Restrictions	Screens saved with one version of the Turbo Pascal Graphix Toolbox are not necessarily compatible with any other version.
See Also	LoadScreen LoadWindow SaveWindow
Example	`SaveScreen('PRETTY.PIC');`

saves the active screen in a file called *PRETTY.PIC*.

SaveWindow [GWINDOW.PAS]

Declaration `procedure SaveWindow(I:integer;FileName:WrkString);`

Usage SaveWindow(I,FileName);

Parameters *I* : index of window to be saved

FileName : file name of window

Function *SaveWindow* creates a file named by *FileName*, and saves window *I* in it. *FileName* can include an extension and a disk drive declaration (e.g., *a:FileName.xxx*). If a file named *FileName* already exists, it is overwritten. The size and position of the window are saved in the file, and are used when the window is loaded with *LoadWindow*, though the position can be changed if positive values are given for the *X* and *Y* coordinates when *LoadWindow* is called.

Remarks Windows saved with different versions of the Turbo Pascal Graphix Toolbox will be compatible.

Restrictions None

See Also LoadScreen
LoadWindow
SaveScreen

Example

```
SaveWindow(15,'MENU.WIN');
```

saves window 15 in a file called *MENU.WIN*.

SaveWindowStack [GWINDOW.PAS]

Declaration	`procedure SaveWindowStack(FileName:WrkString);`
Usage	SaveWindowStack(FileName);
Parameters	*FileName* : file name of window stack
Function	*SaveWindowStack* stores a window stack on a floppy or hard disk. The contents of the stack include all defined and stored windows. The procedure automatically creates two files with extensions, *FileName.STK* (window stack) and *FileName.PTR* (pointer file). For this reason, you should not specify an extension for *FileName*, although a disk drive declaration can be specified. If a file with name *FileName* exists, it is overwritten.
Restrictions	Window stacks saved by different versions of the Turbo Pascal Graphix Toolbox will not necessarily be compatible.
See Also	LoadWindow LoadWindowStack SaveWindow
Example	`SaveWindowStack ('WSTACK');`

saves any windows that are currently stored in the window stack in two disk files, *WSTACK.STK* and *WSTACK.PTR*.

ScalePolygon [GSHELL.PAS]

Declaration

```
procedure ScalePolygon(var A:PlotArray; N:integer;
                       XFactor,YFactor:Float);
```

Usage: ScalePolygon(A,N,XFactor,YFactor);

Parameters:

A : polygon array

N : number of polygon vertices

XFactor : multiplication factor (scaling) in *X* direction

YFactor : multiplication factor (scaling) in *Y* direction

Function: *ScalePolygon* scales the lines that make up a polygon *A* by a proportional amount (*XFactor* and *YFactor*) in both horizontal (*X*) and vertical (*Y*) directions. The *X* coordinate of each of the *N* vertices is multiplied by *XFactor*, and the *Y* coordinate by *YFactor*.

Remarks: When drawing a scaled polygon using *DrawPolygon*, the number of vertices should be given as a negative value, to ensure clipping at window boundaries.

Restrictions: None

See Also: DrawPolygon
RotatePolygon
RotatePolygonAbout
TranslatePolygon

Example

```
ScalePolygon(Image,35,2,0.5);
```

changes the values of the 35 coordinate pairs in *Image* so that the polygon is stretched to twice its former width, and compressed to half its former height.

SelectScreen [GKERNEL.PAS]

Declaration `procedure SelectScreen(I:word);`

Usage SelectScreen(I);

Parameters *I* : displayed or RAM screen

Function *SelectScreen* selects either the displayed or RAM screen for drawing. If *I* is 1, the displayed screen is selected. If *I* is 2, the RAM screen is selected.

Remarks The constant *RamScreenGlb*, defined in GDRIVER.PAS, must be set to TRUE (the default) to enable a RAM screen.

Restrictions Drawing is not visible on the RAM screen unless it is first copied to the displayed screen with *CopyScreen* or *SwapScreen*.

See Also CopyScreen
GetScreen
SwapScreen

Example

```
SelectScreen(1);
```

selects the displayed screen for subsequent drawing.

SelectWindow [GKERNEL.PAS]

Declaration `procedure SelectWindow(I:integer);`

Usage SelectWindow(I);

Parameters *I* : index of selected window ([1..*MaxWindowsGlb*])

Function *SelectWindow* selects a window *I* for drawing. All subsequent drawing and window commands will refer to the selected window.

Remarks If clipping is enabled with the *SetClippingOn* procedure, drawing is limited to the area inside the window.

Restrictions The value of *I* must lie between 1 and the constant *MaxWindowsGlb* (defined in GDRIVER.PAS file).

If a world is to be associated with a window, *SelectWorld* must be called before *SelectWindow*.

See Also DefineWindow
SelectWorld

Example

```
SelectWindow(5);
```

selects window 5 for subsequent operations.

SelectWorld [GKERNEL.PAS]

Declaration `procedure SelectWorld(I:integer);`

Usage SelectWorld(I);

Parameters *I* : index of selected world ([1..*MaxWorldsGlb*])

Function *SelectWorld* selects a world coordinate system, *I*, for the drawing commands that follow. This procedure must be followed by *Select-Window* to associate the world with a window.

Restrictions The value of *I* must lie between 1 and the constant *MaxWorldsGlb* (defined in GDRIVER.PAS file).

See Also DefineWindow
DefineWorld
FindWorld
SelectWindow

Example

```
SelectWorld(3);
SelectWindow(4);
```

selects window 4, with world coordinate system 3, for subsequent operations.

SetAspect [GKERNEL.PAS]

Declaration `procedure SetAspect(Aspect:Float);`

Usage SetAspect(Aspect);

Parameters *Aspect* : aspect ratio for circle

Function *SetAspect* sets the value of the aspect ratio for drawing circles and ellipses. The default value for *Aspect* is the constant *AspectFactor*, defined in the GDRIVER.PAS file. *SetAspect*(1) draws a true circle on any screen.

Remarks The aspect ratio determines the shape of circles and ellipses. Changing the aspect ratio changes how tall a circle is. A machine-dependent constant, *AspectFactor*, specifies a ratio that should give a true circle for a particular physical screen. Drawing the same circle with aspect ratios of *AspectFactor* ÷ 2, *AspectFactor*, and *AspectFactor* × 2 will give three figures of the same width, but each twice as tall as the previous figure.

Restrictions None

See Also AspectFactor (constant)
DrawCartPie
DrawCircleSegment
DrawPolarPie
GetAspect
GetScreenAspect
SetScreenAspect

Example

```
SetAspect(1);
```

causes circles to be correctly proportioned on any screen.

SetBackground [GDRIVER.PAS]

Declaration	`procedure SetBackground(Pattern:byte);`
Usage	SetBackground(Pattern);
Parameters	*Pattern* : bit pattern used for background (0 to 255)
Function	*SetBackground* determines the background pattern of the active window. There are 256 possible patterns, represented by the value of *Pattern*. Shading patterns consist of an 8-bit word repeated across each horizontal line to fill the window. The lowest (1) bit of the pattern is the rightmost pixel on the screen, and the highest (128) is the leftmost.
Remarks	A *Pattern* value of 0 creates a completely black background (which erases the contents of the window), while a value of 255 creates a white background.
Restrictions	None
See Also	DrawSquare
Example	`SetBackground(17);`

fills the active window with the pattern represented by the number 17: 00010001 binary (that is, 1 out of every 4 points are drawn).

SetBackground8 [GDRIVER.PAS]

Declaration `procedure SetBackground8(Pattern:BackgroundArray);`

Usage SetBackground8(Pattern);

Parameters *Pattern* : 8-byte background pattern

Function *SetBackground* fills the active window with the specified bit pattern, *Pattern*. The *BackgroundArray* is an array of 8 bytes. The lowest 3 bits of the screen line number are used to determine which byte of the array to use; i.e., the 0 array element is used on screen lines whose *Y* coordinates divide evenly by 8: for a screen line, *Yi*, array [*Yi* mod 8]. The lowest (1) bit of each byte of pattern is the rightmost, and the highest (128) is the leftmost pixel on the screen.

Restrictions None

See Also SetBackground

Example

```
For I:=0 To 7 Do BackgroundPattern [I]:=I*I;
      SetBackground8(BackgroundPattern);
```

This program fills the active window with the pattern below:

```
   +--------+
 0 |        |
 1 |       *|
 4 |     *  |
 9 |    *  *|
16 |   *    |
25 |   **  *|
36 |  *  *  |
49 |  **   *|
   +--------+
```

SetBackgroundColor [GDRIVER.PAS]

Declaration `procedure SetBackgroundColor(Color:word);`

Usage SetBackgroundColor(Color);

Parameters *Color* : background color

Function *SetBackgroundColor* chooses the background color ("black") from the colors available to your particular graphics card. Its value lies between the constants *MinBackground* and *MaxBackground* (defined in the GDRIVER.PAS file).

Remarks *InitGraphic* and *EnterGraphic* always reset colors to true black and white.

Restrictions For IBM versions, the value of *SetBackgroundColor* must be 0 (true black) for the IBM color graphics adapter and the 3270 PC, or can be between 1 and 15 for the PCjr or Enhanced Graphics Adapter; the value of *SetForegroundColor* can be between 1 and 15. For Hercules, *SetBackgroundColor* and *SetForegroundColor* must both be 0, always black and white (or green or amber depending on the monitor). Changing the colors changes the current display, and may have other system-dependent consequences; see Appendix A for more information.

See Also Appendix A
SetForegroundColor

Example

```
SetBackgroundColor (4);
```

sets the color "black" to whatever color 4 represents for the particular graphics card installed. Any "black" images currently displayed immediately change to color 4.

SetBreakOff [GKERNEL.PAS]

Declaration `procedure SetBreakOff;`

Usage SetBreakOff;

Function *SetBreakOff* turns break mode off. When break mode is enabled with the *SetBreakOn* procedure, system errors cause the program to halt. With break mode off, the program proceeds, and it is up to the programmer to check for errors.

Remarks The default state is break mode on.

GetErrorCode returns the code of the last error, or -1 if no error has occurred since the last call to *GetErrorCode*. If a second error happens before the first is cleared, the first error code is lost. See *Error* for discussion.

Restrictions None

See Also Error
GetErrorCode
SetBreakOn
SetMessageOff
SetMessageOn

Example

```
SetBreakOff;
```

causes the program to continue in the event of a graphics error.

SetBreakOn [GKERNEL.PAS]

Declaration	`procedure SetBreakOn;`
Usage	SetBreakOn;
Function	*SetBreakOn* turns break mode on. When an error occurs, the program halts and the error routine takes control of the program. The program counter value where the error occurred and an error code are displayed if *SetMessageOn* is enabled.
Remarks	The default state is break mode on. To allow a program to continue when an error occurs, *SetBreakOff* must be called.
Restrictions	None
See Also	Error GetErrorCode SetBreakOff SetMessageOff SetMessageOn
Example	`SetBreakOn;` causes graphics errors to abort the program.

SetClippingOff [GKERNEL.PAS]

Declaration	`procedure SetClippingOff;`
Usage	SetClippingOff;
Function	*SetClippingModeOff* turns clipping mode off. All images are drawn in their entirety, regardless of window boundaries.
Remarks	The default state is clipping mode on. *SetClippingOff* causes drawing to take place somewhat faster; however, this procedure should be used with caution, since an attempt to draw outside window boundaries using invalid coordinates can cause a system crash and/or overwriting of program memory.
Restrictions	None
See Also	Clip Clipping SetClippingOn SetWindowModeOff SetWindowModeOn
Example	`SetClippingOff;` allows drawings to spill over the boundaries of the active window.

SetClippingOn [GKERNEL.PAS]

Declaration `procedure SetClippingOn;`

Usage SetClippingOn;

Function *SetClippingOn* turns clipping mode on. If part of a drawing falls outside the boundaries of the active window, it is not drawn.

Remarks The default state is clipping mode on.

Drawing takes place somewhat slower in this mode than with *SetClippingOff*, but it is the safer procedure to use, since drawings are prevented from encroaching on program or data memory.

Restrictions None

See Also Clip
Clipping
SetClippingOff
SetWindowModeOff
SetWindowModeOn

Example

```
SetClippingOn;
```

causes any part of a drawing that strays outside window boundaries to be clipped.

SetColorBlack [GKERNEL.PAS]

Declaration `procedure SetColorBlack;`

Usage SetColorBlack;

Function *SetColorBlack* selects "black" as the current drawing color. All further text and graphics will be drawn in "black" until a call to *SetColorWhite*.

Remarks Default drawing color is white.

"Black" can be any background color supported by your graphics card, except true white; see *SetForegroundColor*.

You may want to use the *SetBackground* procedure to fill a window with a non-black pattern before drawing in "black".

When *SetColorBlack* has been called, the *PointDrawn* function will return TRUE if the specified point is drawn in black.

Restrictions For systems with color graphics cards, the color substituted for "black" cannot be true white.

See Also
DrawPoint
GetColor
PointDrawn
SetBackground
SetBackgroundColor
SetColorWhite
SetForegroundColor

Example

```
SetColorBlack;
```

causes subsequent images to be drawn in "black" (the background color).

SetColorWhite [GKERNEL.PAS]

Declaration `procedure SetColorWhite;`

Usage SetColorWhite;

Function *SetColorWhite* selects "white" as the current drawing color. All further text and graphics will be drawn in "white" until a call to *SetColorBlack*.

Remarks Default drawing color is white.

"White" can be any foreground color supported by your graphics card, except true black; see *SetForegroundColor*.

You may want to use *SetBackground* to fill a window with a non-white pattern before drawing in white.

When *SetColorWhite* has been called, the *PointDrawn* function returns TRUE when the specified point is drawn in white.

Restrictions On systems with color graphics cards, the color represented by "white" cannot be true black.

See Also DrawPoint
GetColor
PointDrawn
SetBackground
SetBackgroundColor
SetColorBlack
SetForegroundColor

Example

```
SetColorWhite;
```

causes subsequent images to be drawn in "white" (the foreground color).

SetForegroundColor [GDRIVER.PAS]

Declaration `procedure SetForegroundColor(Color:word);`

Usage SetForegroundColor(Color);

Parameters *Color* : color of displayed text and graphics

Function *SetForegroundColor* selects the drawing color from the colors available to your particular graphics card. Its value lies between the constants *MinForeground* and *MaxForeground* (defined in GDRIVER.PAS).

Remarks *InitGraphic* and *EnterGraphic* always reset colors to true black and white.

See the discussion under *SetBackgroundColor* for more information.

Restrictions See *SetBackgroundColor*.

See Also SetBackgroundColor
SetColorBlack
SetColorWhite

Example

```
SetForegroundColor(9);
```

sets the color "white" as whatever color 9 represents on the particular graphics card installed. Any "white" images currently displayed immediately change to color 9.

SetHeaderOff [GKERNEL.PAS]

Declaration `procedure SetHeaderOff;`

Usage SetHeaderOff;

Function *SetHeaderOff* suppresses the display of window headers and footers until a call to *SetHeaderOn*. This means that *DrawBorder* will not display any header or footer unless *SetHeaderOn* has been called.

Remarks The default state is header mode off.

Windows currently displayed on the screen are not affected by *SetHeaderOff*.

See *DefineHeader* for how to define headers.

Restrictions None

See Also DefineHeader
DrawBorder
RemoveHeader
SetHeaderOn
SetHeaderToBottom
SetHeaderToTop

Example

```
SetHeaderOff;
```

subsequent calls to *DrawBorder* will not draw a header for any window, even if a header is defined.

SetHeaderOn [GKERNEL.PAS]

Declaration `procedure SetHeaderOn;`

Usage SetHeaderOn;

Function *SetHeaderOn* allows window headers and footers to be displayed when *DrawBorder* is called.

Remarks Default state is header mode off.

SetHeaderOn does not affect windows currently displayed on the screen.

See *DefineHeader* for how to define window headers.

Restrictions None

See Also DefineHeader
DrawBorder
RemoveHeader
SetHeaderOff
SetHeaderToBottom
SetHeaderToTop

Example `SetHeaderOn;`

subsequent calls to *DrawBorder* will draw a header for any window for which a header is defined.

SetHeaderToBottom [GKERNEL.PAS]

Declaration `procedure SetHeaderToBottom;`

Usage SetHeaderToBottom;

Function *SetHeaderToBottom* displays all headers at the bottom edge of windows, i.e., as footers, when *DrawBorder* is called.

Remarks Headers are displayed at the top of windows by default.

This procedure does not affect windows currently displayed on the screen.

See *DefineHeader* for how to define window headers.

Restrictions None

See Also DefineHeader
DrawBorder
RemoveHeader
SetHeaderOff
SetHeaderOn
SetHeaderToTop

Example

```
SetHeaderToBottom;
```

subsequent calls to *DrawBorder* will draw window headers at the bottom of windows.

SetHeaderToTop [GKERNEL.PAS]

Declaration `procedure SetHeaderToTop;`

Usage SetHeaderToTop;

Function *SetHeaderToTop* allows window headers to be drawn at the top edge of windows when *DrawBorder* is called.

Remarks Headers are displayed at the top of windows by default.

SetHeaderToTop does not affect windows currently displayed on the screen.

See *DefineHeader* for how to define window headers.

Restrictions None

See Also DefineHeader
DrawBorder
RemoveHeader
SetHeaderOff
SetHeaderOn
SetHeaderToBottom

Example
```
SetHeaderToTop;
```

subsequent calls to *DrawBorder* will draw window headers at the top of windows.

SetLineStyle [GKERNEL.PAS]

Declaration `procedure SetLineStyle(LS:word);`

Usage SetLineStyle(LS);

Parameters *LS* : one of five possible line styles

Function *SetLineStyle* selects one of five available line styles for drawing lines; custom patterns can also be designed. Patterns consist of eight repeating pixels. The five predefined patterns are

```
0: ****************  (unbroken line)
1: *    *    *    *  (dotted line)
2: *****     *****   (dashed line)
3: ***   *   ***   * (dash-dot-dash-dot)
4: *** *** *** ***   (short dashes)
```

Any integer value larger than 4 is interpreted according to the modulo function; that is, the high-order byte of the integer is discarded, and the remaining 8 bits specify the pattern to be repeated. The lowest bit comes first. Thus, a linestyle of 100 decimal is 01100100 binary, for a linestyle of

```
** *  ** *  ** *  ** *  ** * ...
```

Remarks *GetLineStyle* returns the linestyle as a value of 0 to 4 for the predefined patterns, and 256 + pattern for custom patterns.

Restrictions None

See Also GetLineStyle

Example

```
SetLineStyle(1);
```

sets the line style to pattern 1, a dotted line.

```
SetLineStyle(117);
```

sets the line style to the bit pattern represented by decimal 117, binary 01110101, as follows:

```
*** * * *** * * ...
```

SetMessageOff [GKERNEL.PAS]

Declaration `procedure SetMessageOff;`

Usage SetMessageOff;

Function *SetMessageOff* suppresses the display of complete error messages. However, if break mode is enabled with the *SetBreakOn* procedure, a brief, non-explanatory message is displayed. The following table shows how error messages are handled by *SetMessageOff* and *SetMessageOn*, in conjunction with *SetBreakOn* and *SetBreakOff*.

	SetMessageOn	**SetMessageOff**
SetBreakOn Enabled	Complete error message including traceback displayed; halts	"Graphics error ", proc, code displayed; program halts.
SetBreakOff Enabled	Complete error message displayed on line 24; program continues with no traceback.	No message; program continues.

Remarks The default state is message mode on.

The reason a brief message is displayed with *SetMessageOff* is so that, if you sell a program written with the Turbo Pascal Graphix Toolbox, your end users can provide you with information about the cause of an error.

Restrictions None

See Also Error
GetErrorCode
SetBreakOff
SetBreakOn

Example

```
SetMessageOff;
```

if break mode is off, errors will not cause error messages to be displayed. If break mode is on, only a brief error message is displayed before the program is aborted.

SetMessageOn [GKERNEL.PAS]

Declaration	`procedure SetMessageOn;`
Usage	SetMessageOn;
Function	*SetMessageOn* allows complete error messages to be displayed, whether break mode is enabled or not. See the table under *SetMessageOff* for an explanation of how error messages are handled by the *SetMessage* procedures.
Remarks	Default state is message mode on.
Restrictions	None
See Also	Error GetErrorCode SetBreakOff SetBreakOn SetMessageOff
Example	`SetMessageOn;`

If break mode is off, errors will cause error messages to be displayed on screen line 24. If break mode is on, error messages will include the name of the procedure and the nature of the error, along with a traceback.

NO

SetScreenAspect [GKERNEL.PAS]

Declaration `procedure SetScreenAspect(Aspect:Float);`

Usage SetScreenAspect(Aspect);

Parameters *Aspect* : aspect ratio for circle, in pixels

Function *SetScreenAspect* sets the value of the aspect ratio, in pixels, for drawing circles and ellipses. *SetScreenAspect*(1) makes a circle or ellipse that is equal in pixel width and height.

Remarks This procedure is used for applications in which you need to create a circle or ellipse that is proportional in terms of pixels. Note that such a circle is not necessarily correctly proportioned when viewed on the screen; a certain number of consecutive pixels displayed horizontally is quite a bit shorter in length than the same number vertically. An aspect ratio of about 0.6 often gives a truer circle on the screen. Use *SetAspect* to draw visually proportioned circles on a particular screen.

Restrictions None

See Also AspectFactor (constant)
DrawCartPie
DrawCircleSegment
DrawPolarPie
GetAspect
GetScreenAspect
SetAspect

Example

```
SetScreenAspect(1);
```

causes circles to have the same number of vertical as horizontal pixels.

SetVStep [GKERNEL.PAS]

Declaration	`procedure SetVStep(Step:word);`
Usage	SetVStep(Step);
Parameters	*Step* : number of vertical pixels moved by a window at one time
Function	*SetVStep* specifies the vertical distance, in pixels, that a window moves at one time. *Step* can be any integer value larger than 0. Small *Step* values cause smooth, slower window movement, while larger values cause faster, but somewhat jerkier movement.
Remarks	The default value for *VStep* depends on the resolution produced by the particular graphics card installed in your system. This default value is set by the constant *IVStep* in GDRIVER.PAS. See Appendix A for more information on hardware configurations. If a window is moved a distance that is not a multiple of the current *VStep* value, it is moved by multiples of *VStep* towards its destination, then one final, variable-length *Step* to reach its destination. See *MoveVer*.
Restrictions	The value for *Step* must be a positive integer.
See Also	Appendix A MoveVer
Example	`SetVStep(12);`

causes vertical window movement (with the *MoveVer* procedure) to take place in 12-pixel increments.

SetWindowModeOff [GKERNEL.PAS]

Declaration `procedure SetWindowModeOff;`

Usage SetWindowModeOff;

Function *SetWindowModeOff* allows drawing to take place on the screen, in absolute screen coordinates, rather than in a window. Drawings are not clipped at window boundaries unless clipping is enabled with the *SetClippingOn* procedure.

Remarks Default state is window mode on.

Since no clipping is performed when *SetWindowModeOff* has been called, drawing takes place somewhat faster. However, this procedure should be used with caution, since invalid coordinates can cause drawing to encroach on program memory or crash the system.

Restrictions None

See Also DefineWindow
SelectWindow
SetClippingOff
SetClippingOn
SetWindowModeOn

Example

```
SetWindowModeOff;
```

turns window mode off, so that subsequent coordinates are calculated as screen coordinates, with no clipping at window boundaries.

SetWindowModeOn [GKERNEL.PAS]

Declaration	`procedure SetWindowModeOn;`
Usage	SetWindowModeOn;
Function	*SetWindowModeOn* allows you to draw in a window, in world coordinates. Drawings are clipped at the active window boundaries if clipping is enabled with the *SetClippingOn* procedure.
Remarks	Default state is window mode on. Although drawing takes place somewhat slower with window mode on, *SetWindowModeOn* is the safer procedure to use, since clipping at window boundaries is possible and program memory therefore protected.
Restrictions	None
See Also	DefineWindow SelectWindow SetClippingOff SetClippingOn SetWindowModeOff
Example	`SetWindowModeOn;` turns window mode on, so that world coordinate systems can be used, and drawings can be clipped at window boundaries.

Spline [GSHELL.PAS]

Declaration

```
procedure Spline(A:PlotArray; N:integer; X1,XM:Float;
                 var B:PlotArray; M:integer);
```

Usage Spline(A,N,X1,XM,B,M);

Parameters

A : polygon array (base points)

N : number of base points

X1 : index value from which interpolation begins

ACTUAL

XM : index of value where interpolation ends

B : resultant spline polygon array (to be filled with calculated spline)

M : number of points to calculate in spline array

Function When polygons are plotted with a few data points, the connection of these points sometimes results in a vague, angular representation of the true curve. One way to resolve this problem is to evaluate additional base points to smooth the graph plot. However, the calculation time involved in this method may be prohibitive.

The spline functions use smoothing polynomials to generate additional base points. Spline functions are stable in all parts of the definition interval and, unlike many other polynomials, they do not tend to have strong oscillations.

The *Spline* procedure calculates smoothed curves from corresponding data. The number and density of the interpolated points created by the spline function is arbitrary.

To use the *Spline* procedure, first pass a *PlotArray* and the number of points in the array (*N*). *X1* and *XM* specify the starting and ending points, respectively, for the interpolation. The *PlotArray B* receives the resultant interpolated curve. The calculated base points are evenly spaced between the starting and ending points of the input curve.

The spline function is calculated with the following formula:

$$p_n(x) = y_1 \frac{(x-x_2)\cdots(x-x_n)}{(x_1-x_2)\cdots(x_1-x_n)} + y_2 \frac{(x-x_1)(x-x_3)\cdots(x-x_n)}{(x_2-x_1)(x_2-x_3)\cdots(x_2-x_n)} + \cdots + y_n \frac{(x-x_1)\cdots(x-x_{n-1})}{(x_n-x_1)\cdots(x_n-x_{n-1})}$$

Restrictions For the base points of the interpolation the following conditions apply:

$X1 \geq X2 \qquad XN - 1 \leq XM$

$X2/N - 1$ represents the second/second to the last point of the polygon. The interpolation may only be carried out within that interval.

Example

```
Spline(RoughCurve,10,5.7,213,SmootherCurve,50);
```

interpolates a smoothed 50-point curve from the given 10-point curve, over the X range of 5.7 to 213.

StoreWindow [GWINDOW.PAS]

Declaration `procedure StoreWindow(Window:integer);`

Usage StoreWindow(Window);

Parameters *Window* : index of window to be saved [1..*MaxWindowsGlb*]

Function *StoreWindow* saves a given window in the window stack. The procedure checks the window memory to see if sufficient space is available to store the window. If space is not available, an error occurs and the window is not stored. If a previously stored window and the active window share the same index number, the active window overwrites the stored window.

Remarks Storing a window does not affect the screen display.

Stored windows are dynamically allocated on the heap with Turbo Pascal's *GetMem* and *FreeMem* procedures. Windows are always allocated in multiples of 1K (1024) bytes. Because the *StoreWindow* and *RestoreWindow* procedures use *GetMem* and *FreeMem*, your program must not use the Mark/Release method of memory management.

Turbo Pascal's built-in function, *MaxAvail* can be used to determine whether a window will fit on the stack. *MaxAvail* returns the size of the largest chunk of free memory on the stack. By comparing *MaxAvail* to *WindowSize*, which returns the amount of memory required by a particular window, you can tell if there is sufficient room on the stack for the window; that is,

```
if MaxAvail > WindowSize(i) then ok
```

Restrictions The value for *Window* must lie between 1 and the constant *MaxWindowsGlb* (defined in the GDRIVER.PAS file). If an illegal window number is given for Window, or if the stack is out of space, an error occurs.

See Also RestoreWindow
WindowSize

Example

```
StoreWindow(12);
```

causes window 12 to be copied to the window stack for later retrieval.

SwapScreen [GDRIVER.PAS]

Declaration	`procedure SwapScreen;`
Usage	SwapScreen;
Function	*SwapScreen* exchanges the contents of the displayed screen with the contents of the RAM screen.
Remarks	The active screen is not changed by *SwapScreen*. This means that, if you are drawing on one screen and call *SwapScreen* while you are still drawing, the part of the drawing that is complete is moved to the inactive screen, but subsequent drawing takes place on the active screen.
Restrictions	This procedure can only be used if a RAM screen is allocated, i.e., *RamScreenGlb* is TRUE (defined in GDRIVER.PAS file).
See Also	CopyScreen LoadScreen SaveScreen SelectScreen
Example	`SwapScreen;`

swaps the contents of the displayed and RAM screens.

TextDown [GKERNEL.PAS]

Declaration `function TextDown(TY, Boundary:integer):integer;`

Usage TextDown(TY,Boundary);

Parameters *TY* : *Y* coordinate of given machine-dependent text that is to be within a window

Boundary : desired number of pixels between text and bottom edge of window

Function *TextDown* uses the given *Y* text coordinate, *TY*, and the number of pixels, *Boundary*, that you want to have between the text and the bottom edge of the window, to calculate a *Y* window definition coordinate. The function then returns the *Y* coordinate of the bottom edge of a window that is at least *Boundary* pixels below the bottom edge of text coordinate *TY*. Margin at Top of WINDOW

Remarks Along with *TextLeft*, *TextRight*, and *TextUp*, this function is used to fit and align text within a window. It is particularly useful with the Hercules version of the Turbo Pascal Graphix Toolbox, since Hercules text is defined on 9-pixel boundaries, while windows are defined on 8-pixel boundaries; this 1-pixel offset can cause alignment problems. If you want a uniform space between your text and all four window boundaries, use the *DefineTextWindow* procedure. See Appendix A for more information.

Restrictions None

See Also Appendix A
DefineTextWindow
TextLeft
TextRight
TextUp

Example

```
I:=TextDown(16,2);
```

sets *I* to the *Y* screen coordinate at the bottom of row 16, with a boundary of 2 pixels between the text and the window.

TextLeft [GKERNEL.PAS]

Declaration `function TextLeft(TX, Boundary:integer):integer;`

Usage TextLeft(TX,Boundary);

Parameters
TX : *X* coordinate of given machine-dependent text that is to be inside a window

Boundary : desired number of pixels between text and left edge of window

Function *TextLeft* uses the given *X* text coordinate, *TX*, and the number of pixels, *Boundary*, that you want to have between the text and the left edge of the window, to calculate an *X* window definition coordinate. The function then returns the *X* coordinate of the left edge of a window that is at least *Boundary* pixels to the left of the left edge of text coordinate *TX*. Margin RIGHT EDGE OF WINDOW

Remarks Along with *TextDown*, *TextRight*, and *TextUp*, this function is used to fit and align text within a window. It is particularly useful with the Hercules version of the Turbo Pascal Graphix Toolbox, since Hercules text is defined on 9-pixel boundaries, while windows are defined on 8-pixel boundaries; this 1-pixel offset can create alignment problems. If you want a uniform space between your text and all four window boundaries, use the *DefineTextWindow* procedure. See Appendix A for more information.

Restrictions None

See Also
Appendix A
DefineTextWindow
TextDown
TextRight
TextUp

Example

```
I:=TextLeft(LeftMargin,0);
```

sets *I* to the *X* screen coordinate that corresponds to the left edge of column *LeftMargin*.

TextRight [GKERNEL.PAS]

Declaration `function TextRight(TX,Boundary:integer):integer;`

Usage TextRight(TX,Boundary);

Parameters *TX* : *X* coordinate of given machine-dependent text that is to be inside a window

Boundary : desired number of pixels between text and right edge of window

Function *TextRight* uses the given *X* text coordinate, *TX*, and the number of pixels, *Boundary*, that you want to have between the text and the right edge of the window, to calculate an *X* window definition coordinate. The function then returns the *X* coordinate of the right edge of a window that is at least *Boundary* pixels to the right of the right edge of text coordinate *TX*. Margin at Left Edge

Remarks Along with *TextDown*, *TextLeft* and *TextUp*, this function is used to fit and align text within a window. It is particularly useful with the Hercules version of the Turbo Pascal Graphix Toolbox, since Hercules text is defined on 9-pixel boundaries, while windows are defined on 8-pixel boundaries; this 1-pixel offset can create alignment problems. If you want a uniform space between your text and all four window boundaries, use the *DefineTextWindow* procedure. See Appendix A for more information.

Restrictions None

See Also Appendix A
DefineTextWindow
TextDown
TextLeft
TextUp

Example
```
J:=TextRight(68,1);
```

sets *J* to the *X* screen coordinate that is at least 1 pixel to the right of column 68.

TextUp [GKERNEL.PAS]

Declaration `function TextUp(TY, Boundary:integer):integer;`

Usage TextUp(TY,Boundary);

Parameters *TY* : *Y* coordinate of given machine-dependent text that is to be within a window

Boundary : desired number of pixels between text and top edge of window

Function *TextUp* uses the given *Y* text coordinate, *TY*, and the number of pixels, *Boundary*, that you want to have between the text and the top edge of the window, to calculate a *Y* window definition coordinate. The function then returns the *Y* coordinate of the upper edge of a window that is at least *Boundary* pixels above the top edge of text coordinate *TY*. Margin at bottom edge

Remarks Along with *TextLeft*, *TextRight*, and *TextDown*, this function is used to fit and align text within a window. It is particularly useful with the Hercules version of the Turbo Pascal Graphix Toolbox, since Hercules text is defined on 9-pixel boundaries, while windows are defined on 8-pixel boundaries; this 1-pixel offset can create alignment problems. If you want a uniform space between your text and all four window boundaries, use the *DefineTextWindow* procedure. See Appendix A for more information.

Restrictions None

See Also Appendix A
DefineTextWindow
TextDown
TextLeft
TextRight

Example `U:=TextUp(TopLine,HeaderSize);`

sets *U* to the *Y* screen coordinate that is *HeaderSize* pixels above row *TopLine*.

TranslatePolygon [GSHELL.PAS]

Declaration
```
procedure TranslatePolygon(var A:PlotArray; N:integer;
                           DeltaX,DeltaY:Float);
```

Usage
TranslatePolygon(A,N,DeltaX,DeltaY);

Parameters
A : polygon array

N : number of polygon vertices

DeltaX : displacement in *X* direction

DeltaY : displacement in *Y* direction

Function
TranslatePolygon moves all polygon line endpoints by adding *X* and *Y* displacements, thus moving the entire polygon both vertically by *DeltaX* and horizontally by *DeltaY*.

Remarks
When drawing a translated polygon using *DrawPolygon*, the number of vertices should be passed as a negative value, so that *DrawPolygon* clips the polygon at window boundaries.

Restrictions
None

See Also
DrawPolygon
RotatePolygon
ScalePolygon

Example
```
TranslatePolygon(Image,73,25,-19.8)
```

changes the values of the 73 coordinate pairs in *Image* so that the polygon is moved 25 *X* units to the right, and 19.8 *Y* units towards the top of the screen.

WindowMode [GKERNEL.PAS]

Declaration `function WindowMode:boolean;`

Usage WindowMode;

Function *WindowMode* returns the window status: TRUE if *WindowModeOn* has been called, FALSE if *WindowModeOff* has been called.

Restrictions None

See Also SetWindowModeOff
SetWindowModeOn

Example `B:=WindowMode;`

B is TRUE if window mode is currently enabled.

WindowSize [GWINDOW.PAS]

Declaration `function WindowSize (Nr:integer):word;`

Usage WindowSize(Nr);

Parameters *Nr* : index of window [1..*MaxWindowsGlb*]

Function *WindowSize* calculates the size of a window in bytes. In a window stack operation, this size is compared to the available window stack space to see if there is sufficient room for the window in the stack; if not, an error occurs.

The formula used for this calculation is:

$WindowSize := (Y2 - Y1 + 1)\,(X2 - X1\cancel{)} + 1)$

The value returned is rounded up to the nearest 1,024 to match with the amount of space the window will consume if it is saved on the window stack. [*X1*,*Y1*] are the coordinates of the left upper corner of the window, and [*X2*,*Y2*] are the coordinates of the right lower corner of the window.

Restrictions The value of *Nr* must lie between 1 and the constant *MaxWindowsGlb* (defined in the GDRIVER.PAS file).

See Also ClearWindowStack
RestoreWindow
StoreWindow

Example `I:=WindowSize(3);`

I contains the number of bytes needed to store window 3 in the window stack.

WindowX [GKERNEL.PAS]

Declaration `function WindowX(X:Float):integer;`

Usage WindowX(X);

Parameters *X* : *X* world coordinate

Function *WindowX* translates an *X* world coordinate into an absolute screen coordinate and returns this value.

Restrictions None

See Also DefineWorld
DefineWindow
SelectWindow
SelectWorld

Example `X:=WindowX(X1);`

converts the world coordinate *X1* to a screen coordinate and stores the value in *X*.

WindowY [GKERNEL.PAS]

Declaration `function WindowY(Y:Float):integer;`

Usage WindowY(Y);

Parameters *Y* : *Y* world coordinate

Function *WindowY* translates a *Y* coordinate from world coordinate to absolute screen coordinates, and returns this value.

Restrictions None

Example `Y:=WindowY(Y1);`

converts world coordinate *Y1* to a screen coordinate and stores the value in *Y*.

A P P E N D I X **A**

Hardware Configurations and Compatibility Problems

This section describes three of the hardware configurations that support the Turbo Pascal Graphix Toolbox. Problems or considerations specific to the IBM and Hercules implementations are first discussed separately; a detailed discussion about compatibility between different hardware configurations follows.

Complete information about the constants, types, procedures and functions mentioned in this appendix can be found in Chapter 3.

The IBM Color Graphics Card

The IBM Color Graphics card supports a hardware environment with the following general characteristics:

- Screen is 640 pixels wide by 200 pixels tall.
- Default step (increment) for vertical window movement (as defined in the constant *IVStep*) is 2 pixels.
- A RAM screen is enabled (constant *RamScreenGlb* = TRUE) and is placed in normal RAM.

Constants take the following default values with the IBM card:

AspectFactor	= 0.44	MinForeground	= 0
HardwareGrafBase	= $B800	RamScreenGlb	= TRUE
IVStep	= 2	ScreenSizeGlb	= 8191
MaxBackground	= *	XMaxGlb	= 79
MaxForeground	= 15	XScreenMaxGlb	= 639
MinBackground	= 0	YMaxGlb	= 199

* depends on version

Color

The different IBM versions of the Turbo Pascal Graphix Toolbox allow either one background color, true black (constants *MinBackground* and *MaxBackground* are both 0), or up to fifteen background colors (*MinBackground* = 0, *MaxBackground* = 15); fifteen foreground colors are available (*MinForeground* = 1, *MaxForeground* = 15), except with the PCjr, which allows only black or white for the foreground color. *MaxForeground* is the default value, set both by the *InitGraphic* and *EnterGraphic* procedures. The following table lists the colors for the IBM Color/Graphics Adapter (CGA), the PCjr, the Enhanced Graphics Adapter (EGA), and the 3270 PC.

Foreground Colors

	CGA	**PCjr**	**EGA**	**3270 PC**
0	Black	Black	Black	Black
1	Blue	White	Blue	Blue
2	Green	Black	Green	Green
3	Cyan	White	Cyan	Turquoise
4	Red	Black	Red	Red
5	Magenta	White	Magenta	Pink
6	Brown	Black	Brown	Yellow
7	Light gray	White	Light gray	White
8	Dark gray	Black	Dark gray	Black
9	Light blue	White	Light blue	Blue
10	Light green	Black	Light green	Green
11	Light cyan	White	Light cyan	Turquoise
12	Light red	Black	Light red	Red
13	Light magenta	White	Light magenta	Pink
14	Yellow	Black	Yellow	Yellow
15	White	White	White	White

Background Colors

	CGA	PCjr	EGA	3270 PC
0	Black	Black	Black	Black
1	Black	Blue	Blue	Black
2	Black	Green	Green	Black
3	Black	Cyan	Cyan	Black
4	Black	Red	Red	Black
5	Black	Magenta	Magenta	Black
6	Black	Brown	Brown	Black
7	Black	Light gray	Light gray	Black
8	Black	Dark gray	Dark gray	Black
9	Black	Light blue	Light blue	Black
10	Black	Light green	Light green	Black
11	Black	Light cyan	Light cyan	Black
12	Black	Light red	Light red	Black
13	Black	Light magenta	Light magenta	Black
14	Black	Yellow	Yellow	Black
15	Black	White	White	Black

Text

In addition to the standard 4x6-pixel font used by Turbo Pascal Graphix, the IBM card allows higher quality text characters to be drawn in the normal IBM Color/graphics adapter font. These characters take the form of 8x8-pixel cells, and can only be drawn at *X* and *Y* coordinates that are multiples of 8 pixels. Since windows are also defined on 8-pixel horizontal boundaries, the higher quality text can be aligned exactly with windows.

Text can be moved vertically to any screen position using the window movement procedure *MoveVer*. Unlike the 4x6-pixel font, IBM text is never clipped at window boundaries.

The Hercules Monochrome Graphics Card

The Hercules Monochrome Graphics card supports a hardware environment with the following general characteristics:

- Screen is 720 pixels wide by 350 pixels tall.
- Default step (increment) for vertical window movement is 5 pixels (as specified by the constant *IVStep*).

- A RAM screen is allocated (constant *RamScreenGlb*=TRUE). The RAM screen can be placed in normal RAM (default) or on the Hercules card itself, as determined by the initialized variable *RamScreenInCard* in the GRAPHIX.HGC file. If *RamScreenInCard* is TRUE, the RAM screen is on the Hercules card; if FALSE, it is in normal RAM. If you change *RamScreenInCard* to TRUE, your Hercules card must be placed in the "full" configuration.

Constants take the following values with the Hercules card:

AspectFactor	= −0.6667	RamScreenGlb	= TRUE
HardwareGrafBase	= $B000	*RamScreenInCard	= FALSE
IVStep	= 5	ScreenSizeGlb	= 16383
MaxBackground	= 0	XMaxGlb	= 89
MaxForeground	= 1	XScreenMaxGlb	= 719
MinBackground	= 0	YMaxGlb	= 349
MinForeground	= 1		

* Specific to the Hercules implementation IN GRAFHGC. DVR

Color

The Hercules card does not support color. Neither background nor foreground color can be changed; both *MinBackground* and *MaxBackground* are set to 0 (black) and both *MinForeground* and *MaxForeground* are set to 1 (white).

Text

In addition to the standard 4x6-pixel font used by Turbo Pascal Graphix to draw window headers and footers, the Hercules card allows higher quality text characters to be drawn on the screen in the normal Hercules font. These characters take the form of 9x14 pixel cells, can only be drawn at text coordinates that start at [0,0], and move in steps (increments) of 9 horizontal pixels by 14 vertical pixels. Using the window movement procedures *MoveHor* and *MoveVer*, you can move text to any desired screen location. However, because Hercules horizontal text coordinates are at multiples of 9 pixels, and window definition coordinates are at multiples of 8, care must be taken when attempting to draw text inside a window; the alignment of text with the window may be slightly skewed due to the repeating 1-pixel offset of text.

Unlike the 4x6-pixel Turbo Pascal Graphix font, Hercules text is never clipped at window boundaries.

Special Notes

Though the Hercules card normally has a resolution of 720x348, through special programming, the Hercules version of the Turbo Pascal Graphix Toolbox changes the resolution to 720x352; the last two vertical pixels are ignored by the program, thus giving a resolution of 720x350. There are a few monitors that may not be able to display this higher resolution. If your monitor loses its horizontal hold when you use Turbo Pascal Graphix, you must change two constants in GDRIVER.PAS: *YMaxGlb* should be changed from 349 to 347, and *VRowsGlb* should be changed from $58 to $57. Be sure to change both constants.

With the Hercules card, if a program terminates while in graphics mode, part of the current graphic display will remain on the screen, and part will be erased. This is because MS-DOS does not understand that the computer is in graphics mode, and will try to use the Hercules card as if it were in text mode. To prevent this, you must use the DOS command MODE MONO or run the program HFIX.COM (on the Turbo Pascal Graphix Toolbox distribution disk).

Suppose your program terminates due to an I/O or runtime error. In this case, you will probably want to see the error message, so you should use HFIX.COM, which displays the error message, rather than MODE MONO, which clears the screen. However, part of the error message may scroll off the screen. One way to capture the error message before it disappears is to use the ***Shift-PrtSc*** sequence. DOS will then display the text screen even though there is also a graphics display.

Compatibility Issues

This section discusses the problems involved with writing a program for more than one version of the Turbo Pascal Graphix Toolbox, and offers suggestions for resolving those problems.

Screen Size

Probably the biggest problem involved with writing programs for different Turbo Pascal Graphix versions is that the graphics cards support different screen sizes. This is especially troublesome for drawings that use absolute screen coordinates. You could define a window and display the drawing using world coordinates, which partially resolves the problem; however, the placement of the window itself depends on the resolution of the screen. For instance, on the IBM Color/Graphics Adapter, a window with its upper left corner at [20, 50] and lower right corner at

[60, 150] is a centered window that is approximately 1/4 the size of the screen. On the Hercules card, the same window would be placed slightly further to the left on the screen, and significantly closer to the top, and would take up only about 1/8 of the screen.

One solution to this problem would be to use the global constants *XMaxGlb* and *YMaxGlb* to standardize the placement of the window. The statement

```
DefineWindow(1,XMaxGlb Div 4,YMaxGlb Div 4,XMaxGlb*3 Div 4,
             YMaxGlb*3 Div 4);
```

would define a centered window that takes up approximately 1/4 of the screen, regardless of the actual screen size.

Text Placement

Another potential compatibility problem is text placement. Although the 4x6-pixel text can be placed at any screen coordinates with the *DrawText* procedure, or at any world coordinates with the *DrawTextW* procedure, recreating the same text on different screens is difficult. This is because the size of the characters may also have to be adjusted. The machine-dependent font is correctly proportioned for the graphics card in use, though it can be difficult to place. For example, suppose, on the IBM Color adapter version, that a window is to be defined that will enclose text coordinates [10, 2] through [20, 4]. The following statement shows one way to define that window:

```
DefineWindow(1,(XMaxGlb*10) Div 80,(YMaxGlb*2) Div 25,
             (XMaxGlb*20) Div 80,(YMaxGlb*4) Div 25);
```

This statement is equivalent to

```
DefineWindow(1,(79*10) Div 80,(199*2) Div 25,(79*20) Div 80,
             (199*4) Div 25);
```

or

```
DefineWindow(1,(790) Div 80,(398) Div 25,(1580) Div 80,
             (796) Div 25);
```

or

```
DefineWindow(1,9,15,19,31);
```

The screen coordinates above are (72, 15, 159, 31) (the *X* screen coordinate is greater by 7 because it includes the entire byte at that coordinate).

On the IBM version, text is drawn at every 8 pixels in both directions, so the screen coordinates to use for a window that includes text coordinates [10, 2] through [20, 4] are (80, 16, 167, 39). If (*XMaxGlb* + 1) and (*YMaxGlb* + 1) were used in the first statement,

```
DefineWindow(1,((XMaxGlb+1)*10) Div 80,((YMaxGlb+1)*2) Div 25,
           ((XMaxGlb+1)*20) Div 80,((YMaxGlb+1)*4) Div 25);
```

the resulting window would be at screen coordinates (80, 16, 167, 32), which would align the text more exactly with the window. Adding 7 to the final *Y* coordinate makes it exact:

```
DefineWindow(1,((XMaxGlb+1)*10) Div 80,((YMaxGlb+1)*2) Div 25,
           ((XMaxGlb+1)*20) Div 80,((YMaxGlb+1)*4) Div 25 + 7);
```

However, if the last statement is used on the Hercules card, the final coordinates come out as (88, 28, 183, 63), which is close to the correct (90, 28, 188, 64)—but not close enough. Text drawn in that window would spill over the right and bottom edges of the window. But the correct window (90, 28, 188, 64) is an illegal window! The first *X* coordinate, 90, is not a multiple of 8, and the second, 188, is not 1 less than a multiple of 8.

Because of the complexity involved in choosing a window to fit text, four functions are provided that choose window definition coordinates based on text coordinates. Each function is given a text coordinate and a minimum boundary value. The function returns a window definition coordinate that will contain the given text coordinate and provide a border of at least the boundary pixel value. The border cannot always be exact because of the difference between text coordinates and byte-at-a-time window coordinates on some machines.

The four functions are

```
TextLeft(TX,Boundary:integer):integer;
```

Returns *X* window coordinate that is at least *Boundary* pixels to the left of the left edge of text coordinate *TX*.

```
TextRight(TX,Boundary:integer):integer;
```

Returns *X* window coordinate that is at least *Boundary* pixels to the right of the right edge of text coordinate *TX*.

```
TextUp(TY,Boundary:integer):integer;
```

Returns *Y* window coordinate that is at least *Boundary* pixels above the top edge of text coordinate *TY*.

```
TextDown(TY,Boundary:integer):integer;
```

Returns *Y* window coordinate that is at least *Boundary* pixels below the bottom edge of text coordinate *TX*.

There are two functions for each direction because the font size is not known to the user program, so the addition of the actual width of the character to its upper lefthand corner coordinate must be done by the system.

Returning to the original example, to define that window enclosing text coordinates [10, 2] through [20, 4] and give a border of at least 1 pixel on all sides, we use

```
DefineWindow(1,TextLeft(10,1),TextUp(2,1),TextRight(20,1),TextDown(4,1));
```

In addition to the four functions, the procedure

```
DefineTextWindow(I,Left,Up,Right,Down,Border:integer);
```

can also be used to adjust a window to text coordinates. This procedure is a more convenient way to solve the alignment problem, since all parameters are defined in one routine; however, it is less flexible, since the size of the border between text and window boundaries must be the same for all four directions.

Color

The color capabilities of the various Turbo Pascal Graphix Toolbox versions range from absolute monochrome (black and white only) to a choice of 16 colors each for the foreground and background. It is very difficult to use the color capabilities in a machine-independent way. The range of colors available is known to the user program, but the actual colors associated with the numbers are not. If two different colors are arbitrarily chosen for foreground and background, there is no way to ensure against, for instance, the choice of blue for foreground and aquamarine for background!

In addition, the consequences of changing the current color vary from machine to machine. On some machines, there may be a considerable delay while pixel colors are being changed. On others, the color may be changed by simply reprogramming the display controller to interpret the same bit patterns as different colors.

Also, on some machines, changing both foreground and background to the same color may destroy the graphic image currently being displayed.

Because of these considerations, it is recommended that programs that are to be used with several versions of the Turbo Pascal Graphix Toolbox be written for true black and white.

Speed

The speed of the Turbo Pascal Graphix Toolbox varies widely on different machines. The variance is not simple; from one machine to the next, one operation may be twice as fast, and another be about the same speed. You should therefore make no assumptions about speed or timing when you are writing a program that is to run on several machines.

Premature Termination

On some machines, if a program ends while still in graphics mode, the computer may behave erratically. For instance, if a program written for the Hercules graphics card version ends without a call to *LeaveGraphic*, DOS does not know that the screen is in graphics mode, and acts as if it is in text mode.

Included on the Turbo Pascal Graphix Toolbox distribution disk is a program, HFIX.COM, to be used to reorient your system after a program terminates improperly. You are free to distribute HFIX.COM with any program you write.

A P P E N D I X **B**

Glossary

absolute screen coordinate system: Coordinate system that uses the entire screen area to plot the pixel location of text or graphics; coordinate [0,0] is in the upper left corner of the screen.

absolute value: The value of a positive or negative number when the sign has been removed. For example, the absolute value of both -2 and $+2$ is 2.

active window: The displayed or RAM (virtual) window in which drawing is currently taking place.

active screen: The displayed or virtual screen in which drawing is currently taking place.

aspect ratio: The horizontal-to-vertical ratio of a circle or ellipse. Used by the Turbo Pascal Graphix Toolbox to proportion circles and pie charts.

background: The screen surface and color on which drawing is taking place. See *foreground*.

bar chart: A graph consisting of vertical or horizontal bars with lengths proportioned according to specified quantities.

base point: Any of the points that constitute a graph or curve.

Bezier function: Function that uses an array of control points to construct a parametric, polynomial curve of a predetermined shape.

Cartesian coordinate system: A method used to plot an object's location according to its horizontal-by-vertical position. This position is referenced to horizontal (*X*) and vertical (*Y*) axes.

clipping: Turbo Pascal Graphix Toolbox function that keeps graphic images within window or screen boundaries by preventing any part of the drawing that falls outside the window or screen from being displayed.

control point: Any of the points used to plot a graph. Used by the Turbo Pascal Graphix Toolbox to construct curves.

coordinate system: A method used to plot an object's location according to its horizontal-by-vertical position. See *absolute screen coordinate system* and *world coordinate system.*

displayed screen: The visible screen displayed on your computer monitor. See *RAM screen.*

flow chart: A graphic representation of a sequence of consecutive events or operations. The Turbo Pascal Graphix Toolbox uses a sequence of moving windows to represent a flow chart.

font: Either of two sets of characters used by the Turbo Pascal Graphix Toolbox. Window headers, and text that must be in multiples of 4x6 pixels, are displayed in the standard 4x6-pixel text font. All other text is displayed in a machine-dependent, higher resolution text font—8x8-pixels for the IBM card, 9x14 pixels for the Hercules card, and 8x9 pixels for the Zenith card.

foreground: The color used to display text and draw graphic images. See *background.*

graphics mode: Mode of computer operation in which graphics symbols and drawings are displayed. See *text mode.*

header: A user-defined text label, displayed in the Turbo Pascal Graphix standard 4x6-pixel font, that is placed either at the top or bottom edge of a window.

histogram: A graphic representation of a frequency distribution that takes the form of a bar chart.

inactive screen: The RAM or displayed screen that is not currently being used for drawing.

include directive: Program comment of the form *{$I filename.ext}* that instructs the compiler to read the program contained in *filename.*

interpolation: Method of determining the value of a function that is between known values, using a procedure or algorithm. See *spline function.*

machine-dependent text: Text that corresponds to the font used by the particular graphics card installed in your system. Text is 8x8-pixels for the IBM card, 8x9 pixels for the Zenith card, and 9x14-pixels for the Hercules card. Machine-dependent text is of a higher resolution than the standard, 4x6-pixel text used by the Turbo Pascal Graphix Toolbox to display window headers. See *font*.

modeling: Method used to find the points (and the corresponding function) that will represent a predetermined line, curve, or solid shape. See *Bezier function*.

origin: In any coordinate system, point [0,0], i.e. the point where the coordinate axes intersect.

pie chart: A circular chart used to represent the relative sizes of several quantities that make up a whole unit. The pie chart is divided into sections by radial lines, with each section proportional in angle and area to the quantity it represents.

pixels: Abbreviation for *picture elements*. The tiny dots that together make up a graphics or text screen display. Pixels are the basic units of measure used by coordinate systems to plot the location of screen objects.

polar coordinate system: Method used to plot a pie chart in reference to its radius and the angle of its first segment.

polygon: A figure that encloses a collection of points, possibly (but not necessarily) connected by line segments.

RAM (virtual) screen: A screen that is stored in RAM memory. It is identical in size and shape to the displayed screen, but any drawing that takes place on it is invisible.

resolution: The quality and accuracy of detail of a displayed image. Resolution depends on the number of pixels within a given area of the screen; the more pixels there are, the higher the resolution.

scaling: Ability of the Turbo Pascal Graphix Toolbox to reduce or enlarge an image to fit in a given window according to the world coordinate system specified by the user.

screen coordinate system: See *absolute screen coordinate system*.

spline function: Polynomial function that smooths a curve by calculating and generating additional base points.

step: The increment by which a text character, window, or graphic image moves at one time.

text mode: Computer mode in which only characters are manipulated and displayed. See *graphics mode*.

vertex: The point where the sides of an angle intersect.

virtual screen: See *RAM screen*.

window: An area of the screen specified by the user for drawing. Can range in size between 1 vertical pixel by 8 horizontal pixels and the entire screen.

window definition coordinates: The two sets of *X* and *Y* coordinates that define the upper left and lower right corners of a window. Windows are defined on 8-bit horizontal by 1-bit vertical boundaries, so that each *X* window definition coordinate represents one 8-pixel horizontal unit, and each *Y* coordinate represents one 1-pixel vertical unit.

window stack: RAM area in which windows can be temporarily stored.

world coordinate system: A user-defined coordinate system that is used to scale drawings within a given window. World *X* (horizontal) and *Y* (vertical) coordinates do not necessarily correspond to actual pixel locations, but can be assigned any values that suit the application. A world is enclosed by the *X* (horizontal) and *Y* (vertical) coordinates of the upper left and lower right corners of the drawing area.

zero axes: The horizontal (*X*) and vertical (*Y*) axes used to plot the location of a screen object.

Index

T

U

W

X

Y

Borland Software

BORLAND
INTERNATIONAL

4585 Scotts Valley Drive, Scotts Valley, CA 95066

Available at better dealers nationwide.
To order by credit card, call (800) 255-8008; CA (800) 742-1133;
CANADA (800) 237-1136.

SIDEKICK:® THE DESKTOP ORGANIZER

Whether you're running WordStar,® Lotus,® dBASE,® or any other program, SideKick puts all these desktop accessories at your fingertips—Instantly!

A full-screen WordStar-like Editor to jot down notes and edit files up to 25 pages long.

A Phone Directory for names, addresses, and telephone numbers. Finding a name or a number is a snap.

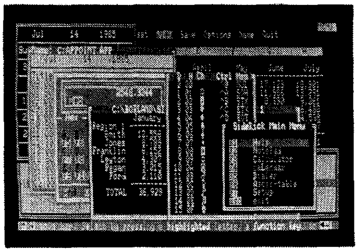
An Autodialer for all your phone calls. It will look up and dial telephone numbers for you. (A modem is required to use this function.)

A Monthly Calendar from 1901 through 2099.

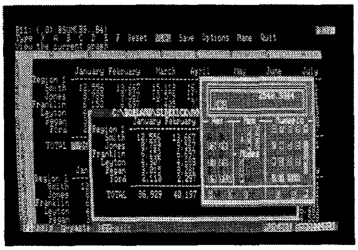
Appointment Calendar to remind you of important meetings and appointments.

A full-featured Calculator ideal for business use. It also performs decimal to hexadecimal to binary conversions.

An ASCII Table for easy reference.



All the SideKick windows stacked up over Lotus 1-2-3.® From bottom to top: SideKick's "Menu Window," ASCII Table, Notepad, Calculator, Appointment Calendar, Monthly Calendar, and Phone Dialer.



Here's SideKick running over Lotus 1-2-3. In the SideKick Notepad you'll notice data that's been imported directly from the Lotus screen. In the upper right you can see the Calculator.

The Critics' Choice

"In a simple, beautiful implementation of WordStar's block copy commands, SideKick can transport all or any part of the display screen (even an area overlaid by the notepad display) to the notepad."
—Charles Petzold, PC MAGAZINE

"SideKick deserves a place in every PC."
—Gary Ray, PC WEEK

"SideKick is by far the best we've seen. It is also the least expensive."
—Ron Mansfield, ENTREPRENEUR

"If you use a PC, get SideKick. You'll soon become dependent on it."
—Jerry Pournelle, BYTE

Suggested Retail Price: $84.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr and true compatibles. PC-DOS (MS-DOS) 2.0 or greater. 128K RAM. One disk drive. A Hayes-compatible modem, IBM PCjr internal modem, or AT&T Modem 4000 is required for the autodialer function.

BORLAND
INTERNATIONAL

If you use an IBM® PC, you need

TURBO Lightning®

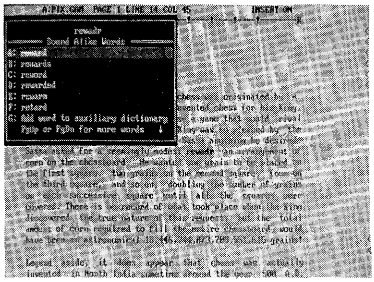
Turbo Lightning teams up with the Random House Concise Word List to check your spelling as you type!

Turbo Lightning, using the 80,000-word Random House Dictionary, checks your spelling *as you type.* If you misspell a word, it alerts you with a "beep." At the touch of a key, Turbo Lightning opens a window on top of your application program and suggests the correct spelling. Just press one key and the misspelled word is instantly replaced with the correct word.

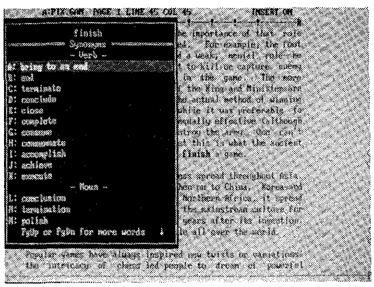
Turbo Lightning works hand-in-hand with the Random House Thesaurus to give you instant access to synonyms

Turbo Lightning lets you choose just the right word from a list of alternates, so you don't say the same thing the same way every time. Once Turbo Lightning opens the Thesaurus window, you see a list of alternate words; select the word you want, press ENTER and your new word will instantly replace the original word. Pure magic!

If you ever write a word, think a word, or say a word, you need Turbo Lightning



The Turbo Lightning Proofreader



The Turbo Lightning Thesaurus

You can teach Turbo Lightning new words

You can *teach* your new Turbo Lightning your name, business associates' names, street names, addresses, correct capitalizations, and any specialized words you use frequently. Teach Turbo Lightning once, and it knows forever.

Turbo Lightning is the engine that powers Borland's Turbo Lightning Library®

Turbo Lightning brings electronic power to the Random House Concise Word List and Random House Thesaurus. They're at your fingertips—even while you're running other programs. Turbo Lightning will also "drive" soon-to-be-released encyclopedias, extended thesauruses, specialized dictionaries, and many other popular reference works. You get a head start with this first volume in the Turbo Lightning Library.

Suggested Retail Price: $99.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr, and true compatibles with 2 floppy disk drives. PC-DOS (MS-DOS) 2.0 or greater. 256K RAM. Hard disk recommended.

BORLAND INTERNATIONAL

REFLEX: THE WORKSHOP™

Includes 22 "instant templates" covering a broad range of business applications (listed below). Also shows you how to customize databases, graphs, crosstabs, and reports. It's an invaluable analytical tool and an important addition to another one of our best sellers, Reflex: The Database Manager.

Fast-start tutorial examples:

Learn Reflex® as you work with practical business applications. The Reflex Workshop Disk supplies databases and reports large enough to illustrate the power and variety of Reflex features. Instructions in each Reflex Workshop chapter take you through a step-by-step analysis of sample data. You then follow simple steps to adapt the files to your own needs.

22 practical business applications:

Workshop's 22 "instant templates" give you a wide range of analytical tools:

Administration
- Scheduling Appointments
- Planning Conference Facilities
- Managing a Project
- Creating a Mailing System
- Managing Employment Applications

Sales and Marketing
- Researching Store Check Inventory
- Tracking Sales Leads
- Summarizing Sales Trends
- Analyzing Trends

Production and Operations
- Summarizing Repair Turnaround
- Tracking Manufacturing Quality Assurance
- Analyzing Product Costs

Accounting and Financial Planning
- Tracking Petty Cash
- Entering Purchase Orders
- Organizing Outgoing Purchase Orders
- Analyzing Accounts Receivable
- Maintaining Letters of Credit
- Reporting Business Expenses
- Managing Debits and Credits
- Examining Leased Inventory Trends
- Tracking Fixed Assets
- Planning Commercial Real Estate Investment

Whether you're a newcomer learning Reflex basics or an experienced "power user" looking for tips, Reflex: The Workshop will help you quickly become an expert database analyst.

Minimum system configuration: IBM PC, AT, and XT, and true compatibles. PC-DOS (MS-DOS) 2.0 or greater. 384K RAM minimum. Requires Reflex: The Database Manager, and IBM Color Graphics Adapter, Hercules Monochrome Graphics Card or equivalent.

BORLAND
INTERNATIONAL

Suggested Retail Price: $69.95 (not copy protected)

TURBO PASCAL®

VERSION 4.0

Turbo Pascal 4.0 has all the features

Turbo Pascal 4.0 has all the features of Turbo Pascal 3.0, *plus* an amazing compilation speed of 27,000 lines per minute,* support for programs larger than 64K, a library of powerful standard units, separate compilation, and much more.

The single-pass, native code compiler offers improved code generation, smart linking to remove unused code from your programs, built-in project management, separate compilation using units, output screen saved in a window, MAP files for use with standard debuggers, a command-line version of the compiler and MAKE utility, and built-in support for 8087/80287/80387 math coprocessors.

All these advanced features, plus the integrated programming environment, online help, and Borland's famous pull-down menus, make Turbo Pascal 4.0 the high-speed, high-performance development tool every programmer hopes for.

Built-in editor

An improved, full-screen editor for editing, compiling, and finding and correcting errors from inside the integrated development environment. Supports 25, 43, and 50 lines per screen, tabs, colors, and new command installation.

Interactive error detection

The compiler instantly locates errors, automatically activates the editor, and shows you the location of the error in the source code.

Pick list

Lets you pick a file from a list of the last eight files loaded into the editor and opens it at the exact spot where you last edited the file. It even remembers your last search string and search options.

Free MicroCalc spreadsheet

A new and improved version of the full-fledged spreadsheet included on your Turbo Pascal disk, absolutely free! You get the complete, revised source code, ready to compile and run.

Compatibility with Turbo Pascal 3.0

A conversion program and compatibility units help you convert all your 3.0 programs to 4.0.

Other Technical Features:

- Several powerful standard units (*System Dos, Crt,* and *Graph*)
- Device-independent graphics support for CGA, MCGA, EGA, VGA, Hercules, AT&T 6300, and IBM 3270 PC
- Extended data types, including *LongInt*
- Optional range- and stack-checking; short-circuit Boolean expression evaluation
- Support for inline statements, inline macros, and powerful assembly language interface
- Faster software-only floating point; toggle switch for 80x87 support including *Single, Double, Extended,* and *Comp* IEEE reals (with numeric coprocessor)
- Automatic execution of initialization and exit code for each unit
- Nested include files up to 8 levels deep, including main module and units
- Operating system calls and interrupts
- Interrupt procedure support for ISRs
- Variable and value typecasting
- Shell to DOS transfer

Minimum system requirements: For the IBM PS/2™ and the IBM® and Compaq® families of personal computers and all 100% compatibles. Integrated environment: 384K; command line: 256K; one floppy drive.
*Run on an 8MHz IBM AT

Suggested retail price $99.95 (not copy protected)

 BOR 0506

TURBO PASCAL®

OOLBOXES AND DEVELOPER'S LIBRARY

n unsurpassed collection of TURBO ASCAL TOOLS that make you the xpert, now upgraded to Version 4.0!

urbo Pascal Tutor:

For both the novice programmer and the profes-onal. Everything you need to write a simple pro-am or handle advanced concepts like using sembly language routines with your Turbo Pascal ograms. The programmer's guide covers the fine ints of Turbo Pascal programming with lots of amples; and on accompanying disk gives you all e source code. A real education for just ***$69.95***!

urbo Pascal Editor Toolbox:

Everything you need to build your own custom xt editor or word processor including easy-to-stall modules, source code and plenty of know-w. Includes all the popular features like word-ap, auto indent, find/replace. Just ***$99.95***!

urbo Pascal Database Toolbox:

A complete library of Pascal procedures that let u sort and search your data and build powerful plications. Includes Turbo Access files that use + trees to organize and search your data, and rbo Sort to sort it. GINST even gets your pro-ams up and running on other terminals! Includes free database that you can use as is or modify to it your needs. Just ***$99.95***!

Turbo Pascal Graphix Toolbox:

Gives you all the high-resolution graphics and graphic window management capabilities you need, with tools to draw and hatch pie charts, bar charts, circles, rectangles and a full range of geometric shapes. Save and restore graphic images to and from disk, plot precise curves, and create anima-tion.* All for just ***$99.95***!

Turbo Pascal GameWorks:

Secrets and strategies of the masters with easy-to-understand examples that teach you how to quickly create your own computer games using Turbo Pascal. For instant excitement, play the three great computer games included on disk—Turbo Chess, Turbo Bridge and Turbo Go-Moku. They're all compiled and ready to run. Just ***$99.95***!

Turbo Pascal Numerical Methods Toolbox:

All the state-of-the-art applied mathematical tools you'll ever need. A collection of Turbo Pascal mathematical routines and programs and ten inde-pendent modules that you can easily adapt to dif-ferent programs. Gives you the kind of mathemati-cal routines IMSL® and NAG libraries provide for FORTRAN. Complete with sample programs and source code for each module. All for just ***$99.95***!

Buy them separately or get The Developer's Library, which includes all six, for just $395 suggested retail price! Not copy protected!

System Requirements: For the IBM PS/2™ and the IBM® and Compaq® families of personal computers and all 100% compatibles.
Operating System: PC-DOS (MS-DOS) 2.0 or later.
**Turbo Pascal Graphix Toolbox* also requires one of the following graphics adapters: CGA, EGA, Hercules, or IBM 3270.

STEP-BY-STEP TUTORIAL AND DEMO PROG WITH SOURCE CODE INCLU

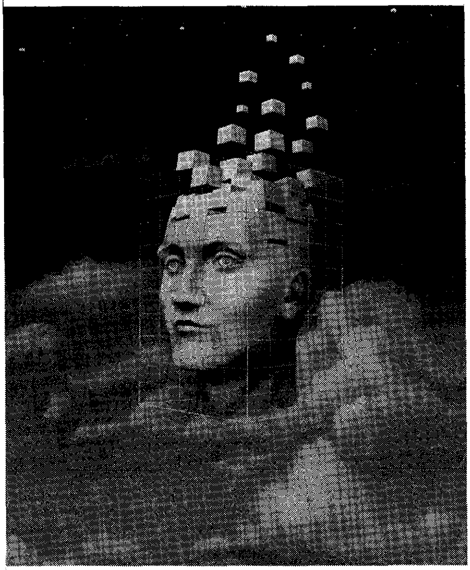
TURBO PROLOG™

the natural language of Artificial Intelligence

Turbo Prolog brings fifth-generation supercomputer power to your IBM®PC!

Turbo Prolog takes programming into a new, natural, and logical environment

With Turbo Prolog, because of its natural, logical approach, both people new to programming *and* professional programmers can build powerful applications such as expert systems, customized knowledge bases, natural language interfaces, and smart information management systems.



Turbo Prolog is a *declarative* language which uses deductive reasoning to solve programming problems.

Turbo Prolog provides a fully integrated programming environment like Borland's Turbo Pascal,® the *de facto* worldwide standard.

You get the complete Turbo Prolog programming system

You get the 200-page manual you're holding, software that includes the lightning-fast Turbo Prolog six-pass compiler and interactive editor, and the free GeoBase natural query language database, which includes commented source code on disk, ready to compile. (GeoBase is a complete database designec and developed around U.S. geography. You can modify it or use it "as is.")

Turbo Prolog's development system includes:

- ☐ A complete Prolog compiler that is a variation of the Clocksin and Mellish Edinburgh standard Prolog.
- ☐ A full-screen interactive editor.
- ☐ Support for both graphic and text windows.
- ☐ All the tools that let you build your own expert systems and **AI** applications with unprecedented ease.

Minimum system configuration: IBM PC, XT, AT, Portable, 3270, PCjr and true compatibles. PC-DOS (MS-DOS) 2.0 or later. 384K RAM minimum.

Suggested Retail Price: $99.95
(not copy protected)

BORLAND
INTERNATIONAL

TURBO PROLOG™ TOOLBOX

Enhances Turbo Prolog with more than 80 tools and over 8,000 lines of source code

Turbo Prolog, the natural language of Artificial Intelligence, is the most popular AI package in the world with more than 100,000 users. Our new Turbo Prolog Toolbox extends its possibilities.

The Turbo Prolog Toolbox enhances Turbo Prolog—our 5th-generation computer programming language that brings supercomputer power to your IBM PC and compatibles—with its more than 80 tools and over 8,000 lines of source code that can be incorporated into your programs, quite easily.

Turbo Prolog Toolbox features include:

- ☑ Business graphics generation: boxes, circles, ellipses, bar charts, pie charts, scaled graphics
- ☑ Complete communications package: supports XModem protocol
- ☑ File transfers from Reflex,® dBASE III,® Lotus 1-2-3,® Symphony®
- ☑ A unique parser generator: construct your own compiler or query language
- ☑ Sophisticated user-interface design tools
- ☑ 40 example programs
- ☑ Easy-to-use screen editor: design your screen layout and I/O
- ☑ Calculated fields definition
- ☑ Over 8,000 lines of source code you can incorporate into your own programs

Suggested Retail Price: $99.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT or true compatibles. PC-DOS (MS-DOS) 2.0 or later. Requires Turbo Prolog 1.10 or higher. Dual-floppy disk drive or hard disk. 512K.

BORLAND INTERNATIONAL

TURBO BASIC® DATABASE TOOLBOX™

With the Turbo Basic Database Toolbox you can build your own powerful, professional-quality database programs. And like all other Borland Toolboxes, it's advanced enough for professional programmers yet easy enough for beginners.

Three ready-to-use modules

The Toolbox enhances your programming with three problem-solving modules:

Turbo Access quickly locates, inserts, or deletes records in a database using B+ trees—the fastest method for finding and retrieving database information. (Source code is included.)

Turbo Sort uses the *Quicksort* method to sort data on single items or on multiple keys. Features virtual memory management for sorting large data files. (Commented source code is on disk.)

TRAINER is a demonstration program that graphically displays how B+ trees work. You can key in sample records and see a visual index of B+ trees being built.

Free sample database

Included is a free sample database with source code. Just compile it, and it's ready to go to work for you—you can use it as is or customize it. You can search the database by keywords or numbers, update records, or add and delete them, as needed.

Saves you time and money

If you're a professional programmer writing software for databases or other applications where search-and-sort capabilities are important, we can save you time and money. Instead of writing the same tedious but essential routines over and over again, you can simply include any of the Toolbox's modules in your own compiled programs.

Technical Features

- ☑ Maximum number of files open: 15 files, or 7 data sets
- ☑ Maximum file size: 32 Mb
- ☑ Maximum record size: 32K
- ☑ Maximum number of records: +2 billion
- ☑ Maximum field size: 32K
- ☑ Maximum key size: 128 bytes
- ☑ Maximum number of keys: +2 billion

Suggested Retail Price: $99.95 (not copy protected)

Minimum system requirements: For the IBM PS/2 and the IBM® and Compaq® families of personal computers and all 100% compatibles, running Turbo Basic 1.0. PC-DOS (MS-DOS®) 2.0 or later. Memory: 640K.

BORLAND
INTERNATIONAL

TURBO C®

Includes free MicroCalc spreadsheet with source code

A complete interactive development environment

With Turbo C, you can expect what only Borland delivers: Quality, Speed, Power and Price. And with its compilation speed of more than 7000 lines a minute, Turbo C makes everything else look like an exercise in slow motion.

Turbo C: The C compiler for both amateurs and professionals
If you're just beginning and you've "kinda wanted to learn C," now's your chance to do it the easy way. Turbo C's got everything to get you going. If you're already programming in C, switching to Turbo C will considerably increase your productivity and help make your programs both smalier and faster.

Turbo C: a complete interactive development environment
Like Turbo Pascal® and Turbo Prolog,™ Turbo C comes with an interactive editor that will show you syntax errors right in your source code. Developing, debugging, and running a Turbo C program is a snap!

Technical Specifications

- ☑ Compiler: One-pass compiler generating native in-line code, linkable object modules and assembler. The object module format is compatible with the PC-DOS linker. Supports small, medium, compact, large, and huge memory model libraries. Can mix models with near and far pointers. Includes floating point emulator (utilizes 8087/80287 if installed).
- ☑ Interactive Editor: The system includes a powerful, interactive full-screen text editor. If the compiler detects an error, the editor automatically positions the cursor appropriately in the source code.
- ☑ Development Environment: A powerful "Make" is included so that managing Turbo C program development is easy. Borland's fast "Turbo Linker" is also included. Also includes pull-down menus and windows. Can run from the environment or generate an executable file.
- ☑ Links with relocatable object modules created using Borland's Turbo Prolog into a single program.
- ☑ ANSI C compatible.
- ☑ Start-up routine source code included.
- ☑ Both command line and integrated environment versions included.

"Sieve" benchmark (25 iterations)

	Turbo C	*Microsoft® C*	*Lattice C*
Compile time	***3.89***	*16.37*	*13.90*
Compile and link time	***9.94***	*29.06*	*27.79*
Execution time	***5.77***	*9.51*	*13.79*
Object code size	***274***	*297*	*301*
Price	***$99.95***	*$450.00*	*$500.00*

Benchmark run on a 6 Mhz IBM AT using Turbo C version 1.0 and the Turbo Linker version 1.0; Microsoft C version 4.0 and the MS overlay linker version 3.51; Lattice C version 3.1 and the MS object linker version 3.05.

Suggested Retail Price: $99.95* (not copy protected) *Introductory offer good through July 1, 1987.

Minimum system configuration: IBM PC, XT, AT and true compatibles. PC-DOS (MS-DOS) 2.0 or later. One floppy drive. 320K.

BORLAND INTERNATIONAL

BOR 0243

SIDEKICK®: THE DESKTOP ORGANIZER Release 2.0

Macintosh™

The most complete and comprehensive collection of desk accessories available for your Macintosh!

Thousands of users already know that SideKick is the best collection of desk accessories available for the Macintosh. With our new Release 2.0, the best just got better.

We've just added two powerful high-performance tools to SideKick—Outlook™: The Outliner and MacPlan™: The Spreadsheet. They work in perfect harmony with each other and *while* you run other programs!

Outlook: The Outliner

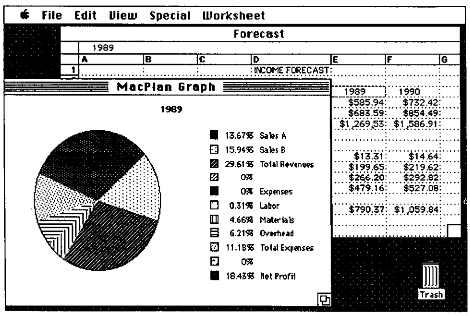
- It's the desk accessory with more power than a stand-alone outliner
- A great desktop publishing tool, Outlook lets you incorporate both text and graphics into your outlines
- Works hand-in-hand with MacPlan
- Allows you to work on several outlines at the same time

MacPlan: The Spreadsheet

- Integrates spreadsheets and graphs
- Does both formulas and straight numbers
- Graph types include bar charts, stacked bar charts, pie charts and line graphs
- Includes 12 example templates free!
- Pastes graphics and data right into Outlook creating professional memos and reports, complete with headers and footers.

SideKick: The Desktop Organizer, Release 2.0 now includes

- ☑ Outlook: The Outliner
- ☑ MacPlan: The Spreadsheet
- ☑ Mini word processor
- ☑ Calendar
- ☑ PhoneLog
- ☑ Analog clock
- ☑ Alarm system
- ☑ Calculator
- ☑ Report generator
- ☑ Telecommunications (new version now supports XModem file transfer protocol)



MacPlan does both spreadsheets and business graphs. Paste them into your Outlook files and generate professional reports.

Suggested Retail Price: $99.95 (not copy protected)

Minimum system configurations: Macintosh 512K or Macintosh Plus with one disk drive. One 800K or two 400K drives are recommended. With one 400K drive, a limited number of desk accessories will be installable per disk.

BORLAND INTERNATIONAL

BOR 0069D

REFLEX® PLUS: THE DATABASE MANAGER
Macintosh™

All the Power & Flexibility of a Relational Database Made Easy! Reflex Plus: The Database Manager is the first relational database that's easy to learn, powerful, and aimed at *your* needs. Reflex Plus is not a mere file organizer, nor is it a monstrously complicated behemoth aimed solely at consultants. Reflex Plus is the only relational database aimed at your needs and time constraints.

Reflex Plus accomplishes this by taking full advantage of the Macintosh's superior graphic ability while still giving users what they want: unlimited flexibility in creating databases, accessing data, and producing reports.

What puts the plus into Reflex Plus?

Borland listens to its customers and has added the most-asked-for features and improvements to Reflex Plus.

High-powered features of Reflex Plus:

- ☑ Multiple entry forms for the same database.
- ☑ Entry for more than one database in a single entry form.
- ☑ Your choice of having an entry form that shows one record at a time, or one that shows all the records at once.
- ☑ Calculated fields in entry forms.
- ☑ Display-only fields.
- ☑ Default (but editable) fields.
- ☑ New functions like GROUPBY, which lets you easily show records grouped by values in common.
- ☑ A selection of useful templates.
- ☑ Larger record size. (You can now choose record sizes of 1000, 2000, or 4000 characters.)

Check out these Reflex Plus features:

- ☑ Visual database design.
- ☑ A "what you see is what you get" design capability both for entry forms and reports.
- ☑ Compatible with all Macintoshes with at least 512K, including the SE™, and Macintosh II.™

The heart of Reflex Plus is in its special functions with which you create formulas. With over 50 function words to choose from, you are given all the power of programming without struggling with complex syntax. Reflex Plus functions are straightforward and can handle all types of data.

Armed with these functions, you create formulas that sort, search, calculate, quantify, qualify—you name it. And if you don't feel up to writing the formula yourself, Reflex Plus will do it for you. Using the FormulaBuild dialog box, you can master even the most complicated formula.

Display grouped data. Reflex Plus gives you unlimited flexibility when you want to display your data grouped in meaningful ways.

Flexible entry forms. Most databases have a data entry form, and that's that. Reflex Plus lets you design your own (but if you don't want to bother, Reflex Plus will make one for you). Here are just some of the options available in your entry forms:

- ☑ View all records at once.
- ☑ View one record at a time.
- ☑ Enter data into many databases at once.
- ☑ Use calculated fields.
- ☑ Default values in fields, display-only values, and lots more.

Convenience and Ease

- ☑ ***Preset entry forms.*** Let Reflex Plus create an entry form for you.
- ☑ ***Preset reports.*** Let Reflex Plus create a table-style report for you.
- ☑ ***Paste Formula command.*** Let Reflex Plus guide you through the steps of creating formulas for power searching and data manipulation.
- ☑ ***On-line help facility.*** Reflex Plus has an extensive on-screen, context-sensitive help feature.
- ☑ ***Paste Choice command.*** This command lets you paste in fields that duplicate all the attributes of another field. A great time saver. The command also lets you build formulas by pointing and clicking.
- ☑ ***Auto-save.*** You'll never lose data again with Reflex Plus's auto-save feature.

Database specifications: Maximum single field length: 4072 bytes. Maximum fields per record: 254. Maximum record length: 4080 bytes. Maximum records per file: limited only by disk capacity. Maximum number of linked database files: 200. Maximum number of open windows: 15. Maximum number of files that can be used by a report: no limit.

Suggested Retail Price: $279.00 (not copy protected)

Minimum system requirements: Runs on any Macintosh with at least 512K memory. Minimum setup is one 800K (double-sided) disk drive or two 400K (single-sided) drives. Works with the Hierarchical File System, Switcher, and most hard disks. Supports printing on the ImageWriter and the LaserWriter.

TURBO PASCAL® MACINTOSH™

The ultimate Pascal development environment

Borland's new Turbo Pascal for the Mac is so incredibly fast that it can compile 1,420 lines of source code in the 7.1 seconds it took you to read this!

And reading the rest of this takes about *5 minutes*, which is plenty of time for Turbo Pascal for the Mac to compile at least *60,000 more lines* of source code!

Turbo Pascal for the Mac does both Windows and "Units"
The *separate* compilation of routines offered by Turbo Pascal for the Mac creates modules called "Units," which can be linked to any Turbo Pascal program. This "modular pathway" gives you "pieces" which can then be integrated into larger programs. You get a more efficient use of memory and a reduction in the time it takes to develop large programs.

Turbo Pascal for the Mac is so compatible with Lisa® that they should be living together
Routines from Macintosh Programmer's Workshop Pascal and Inside Macintosh can be compiled and run with only the subtlest changes. Turbo Pascal for the Mac is also compatible with the Hierarchical File System of the Macintosh.

The 27-second Guide to Turbo Pascal for the Mac

- Compilation speed of more than 12,000 lines per minute
- "Unit" structure lets you create programs in modular form
- Multiple editing windows—up to 8 at once
- Compilation options include compiling to disk or memory, or compile and run
- No need to switch between programs to compile or run a program
- Streamlined development and debugging
- Compatibility with Macintosh Programmer's Workshop Pascal (with minimal changes)
- Compatibility with Hierarchical File System of your Mac
- Ability to define default volume and folder names used in compiler directives
- Search and change features in the editor speed up and simplify alteration of routines
- Ability to use all available Macintosh memory without limit
- "Units" included to call all the routines provided by Macintosh Toolbox

3 MacWinners from Borland!
First there was SideKick for the Mac, then Reflex for the Mac, and now Turbo Pascal for the Mac™!

Suggested Retail Price: $99.95* (not copy protected)

*Introductory price expires July 1, 1987

Minimum system configuration: Macintosh 512K or Macintosh Plus with one disk drive.

BORLAND INTERNATIONAL

TURBO PASCAL®
TUTOR

From the folks who created Turbo Pascal. Borland's new Turbo Pascal Tutor is everything you need to start programming in Turbo Pascal on the Macintosh!™ It takes you from the bare basics to advanced programming in a simple, easy-to-understand fashion.

No gimmicks. It's all here.

The manual, the Tutor application, and 30 sample programs provide a step-by-step tutorial in three phases: programming in Pascal, programming on the Macintosh, and programming in Turbo Pascal on the Macintosh. Here's how the manual is set up:

Turbo Pascal for the Absolute Novice
delivers the basics—a concise history of Pascal, key terminology, your first program.

A Programmer's Guide to Turbo Pascal
covers Pascal specifics—program structure, procedures and functions, arrays, strings, and so on. We've also included Turbo Typist, a textbook sample program.

Advanced Programming
takes you a step higher into stacks, queues, binary trees, linked structures, writing large programs, and more.

Using the Power of the Macintosh
discusses the revolutionary hardware and software features of this machine. It introduces the 600-plus utility routines in the Apple Toolbox.

Programming the Macintosh in Turbo Pascal
shows you how to create true Macintosh programs that use graphics, pull-down menus, dialog boxes, and so on. Finally, MacTypist, a complete stand-alone application featuring animated graphics, builds on Turbo Typist and demonstrates what you can do with all the knowledge you've just acquired.

The disk contains the source code for all the sample programs, including Turbo Typist, MacTypist, and Turbo Tutor. The Tutor's split screen lets you run a procedure and view its source code simultaneously. After running it, you can take a test on the procedure. If you're stuck for an answer, a Hint option steers you in the right direction.

Macintosh topics included are

- ☑ memory management
- ☑ resources and resource files
- ☑ QuickDraw
- ☑ events
- ☑ windows
- ☑ controls
- ☑ menus
- ☑ desk accessory support
- ☑ dialogs
- ☑ File Manager
- ☑ debugging

Suggested Retail Price: $69.95

Minimum system requirements: Any Macintosh with at least 512K of RAM. Requires Turbo Pascal.

BORLAND
INTERNATIONAL

EUREKA: THE SOLVER™

If you're a scientist, engineer, financial analyst, student, teacher, or any other professional working with equations, Eureka: The Solver can do your Algebra, Trigonometry and Calculus problems in a snap.

Eureka also handles maximization and minimization problems, plots functions, generates reports, and saves an incredible amount of time. Even if you're not a computer specialist, Eureka can help you solve your real-world mathematical problems fast, without having to learn numerical approximation techniques. Eureka is easy to learn and easy to use—as simple as a hand-held calculator.

X + exp(X) = 10 solved instantly instead of eventually!

Imagine you have to solve for X, where X + exp(X) = 10, and you don't have Eureka: The Solver. What you do have is a problem, because it's going to take a lot of time guessing at X. With Eureka, there's no guessing, no dancing in the dark—you get the right answer, right now. (PS: X = 2.0705799, and Eureka solved that one in less than 5 seconds!)

How to use Eureka: The Solver

It's easy.

1. Enter your equation into a problem text window
2. Select the "Solve" command
3. Look at the answer
4. You're done

You can then tell Eureka to:

- Verify the solutions
- Draw a graph
- Zoom in on interesting areas of the graph
- Generate a report and send the output to your printer or disk file
- Or all of the above

Some of Eureka's key features

You can key in:

- ☑ A formula or formulas
- ☑ A series of equations—and solve for all variables
- ☑ Constraints (like X must be < or = 2)
- ☑ Functions to plot
- ☑ Unit conversions
- ☑ Maximization and minimization problems
- ☑ Interest Rate/Present Value calculations
- ☑ Variables we call "What happens?," like "What happens if I change this variable to 21 and that variable to 27?"

Eureka: The Solver includes:

- ☑ Calculator+ desk accessory
- ☑ Powerful financial functions
- ☑ Built-in and user-defined functions
- ☑ Reports: generate and save them as MacWrite™ files—complete with graphs and lists—or as Text Only files
- ☑ Polynomial root finder
- ☑ Inequality constraints
- ☑ Logging: keep an up-to-the-minute record of your work
- ☑ Macintosh™ text editor
- ☑ On-screen Help system

Suggested Retail Price: $195.00 (not copy protected)

Minimum system configuration: Macintosh 512K, Macintosh Plus, SE, or II with one 800K disk drive or two 400K disk drives.

BORLAND
INTERNATIONAL

BOR 0415

TURBO PASCAL TOOLBOX™
NUMERICAL METHODS

Turbo Pascal Numerical Methods Toolbox for the Macintosh implements the latest high-level mathematical methods to solve common scientific and engineering problems. Fast.

So every time you need to calculate an integral, work with Fourier transforms, or incorporate any of the classical numerical analysis tools into your programs, you don't have to reinvent the wheel, because the Numerical Methods Toolbox is a complete collection of Turbo Pascal routines and programs that gives you applied state-of-the-art math tools. It also includes two graphics demo programs that use least-square and Fast Fourier Transform routines to give you the picture along with the numbers.

The Turbo Pascal Numerical Methods Toolbox is a must if you're involved with any type of scientific or engineering computing on the Macintosh. Because it comes with complete source code, you have total control of your application at all times.

What Numerical Methods Toolbox will do for you:

- Find solutions to equations
- Interpolations
- Calculus: numerical derivatives and integrals
- Matrix operations: inversions, determinants, and eigenvalues
- Differential equations
- Least-squares approximations
- Fourier transforms
- Graphics

Five free ways to look at Least-Squares Fit!

As well as a free demo of Fast Fourier Transforms, you also get the Least-Squares Fit in five different forms—which gives you five different methods of fitting curves to a collection of data points. You instantly get the picture! The five different forms are

1. Power
2. Exponential
3. Logarithm
4. 5-term Fourier
5. 5-term Poynomial

They're all ready to compile and run as is.

Suggested Retail Price: $99.95 (not copy protected)

Minimum system requirements: Macintosh 512K, Macintosh Plus, SE, or II, with one 800K disk drive (or two 400K).

BORLAND
INTERNATIONAL

BOR 0419

Borland Software ORDER TODAY

BORLAND
INTERNATIONAL

4585 Scotts Valley Drive Scotts Valley, California 95066

To Order By Credit Card, Call (800) 255-8008

In California call (800) 742-1133

In Canada call (800) 237-1136

TURBO PASCAL® GRAPHIX TOOLBOX

A Library of Graphics Routines for Use with Turbo Pascal

IBM® VERSION

PC, XT,® AT,® PS/2™ & True Compatibles

High-resolution graphics for your IBM® PS/2™, PC, AT,® XT,® PC jr® and true compatibles. Comes complete with graphics window management.

Even if you're new to Turbo Pascal programming, the Turbo Pascal Graphix Toolbox will get you started immediately. It's a collection of tools that will get you right into the fascinating world of high-resolution business graphics, including graphics window management. You get immediate, satisfying results. And we keep Royalty out of American business because you don't pay any—even if you distribute your own compiled programs that include all or part of the Turbo Pascal Graphix Toolbox procedures.

The Toolbox Includes

- Commented source code on disk.
- Tools for drawing simple graphics.
- Tools for drawing complex graphics, including curves with optional smoothing.
- Routines that let you store and restore graphic images to and from disk.
- Tools allowing you to send screen images to Epson®-compatible printers.
- Full graphics window management
- Two different font styles for graphic labeling
- Choice of line-drawing styles
- Routines that will let you quickly plot functions and model experimental data.
- Routines that are structured into Pascal units so you don't have to recompile the toolbox code everytime you use it.

"While most people only talk about low-cost personal computer software, Borland has been doing something about it. And Borland provides good technical support as part of the price."

John Markov & Paul Freiberger, syndicated columnists.

If you ever plan to create Turbo Pascal programs that make use of business graphics or scientific graphics, you need the Turbo Pascal Graphix Toolbox.

Minimum system requirements: For the IBM PS/2™, and the IBM® and Compaq® families of personal computers and all 100% compatibles. Turbo Pascal 4.0 or later. 256K RAM minimum. Two disk drives and an IBM Color Graphics Adapter (CGA), IBM Enhanced Graphics Adapter (EGA), IBM 3270 PC, ATT 6300, or Hercules Graphics Card or compatible.

BOR 0455

BORLAND INTERNATIONAL

*4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CA 95066*

*Les Postillons des Bruyeres - 65, rue de la Garenne
92318 SEVRES CEDEX - FRANCE*

*One Great Cumberland Place
London W1H 7AL - United Kingdom*

ISBN 0-87524-174-3